Internet−Scale Information Monitoring: A Continual Query Approach

A Thesis
Presented to
The Academic Faculty

by

Wei Tang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
November 2003

Internet−Scale Information Monitoring: A Continual Query Approach

Approved by:

Ling Liu, Committee Chair
November 25, 2003

Ling Liu, Advisor
November 25, 2003

Constantinos Dovrolis
November 25, 2003

Leo Mark
November 25, 2003

Edward Omiecinski
November 25, 2003

Calton Pu
November 25, 2003

Thomas Potok
November 25, 2003

Date Approved: November 25, 2003

To my dear parents and Weiling

ACKNOWLEDGEMENTS

My PhD study is a long journey. The journey is not only mental, but also physical as well as geographical. It preluded at the University of Alberta in Canada, where I was directed by Dr. Ling Liu for my Master's study. The journey continued in Oregon Graduate Institute where started my PhD pursuit under Dr. Ling Liu's guidance in 1998. One year later, two professors (Dr. Calton Pu and Dr. Ling Liu) led the my project group to the southern city of Atlanta and settled down in Georgia Tech. As I recall the journey along the years, there are many people I am indebted to.

First of all, I want to express my hearty gratitude to my PhD advisor, Dr. Liu, without whom this thesis would not have seen the light of day. She has not only guided me through technical development in research but also taught me the important values in life. My achievement owes much to her selfless help and ever-lasting encouragement and support.

I would also like to thank my thesis committee (Dr. Constantinos Dovrolis, Dr. Leo Mark, Dr. Edward Omiecinski, Dr. Calton Pu, and Dr. Thomas Potok) for participating in my thesis defense and giving their insightful comments on improving the thesis work. I want to give special thanks to Dr. Calton Pu for his help through our research meetings. I appreciate his unique style of guidance and his sharp vision in computer science research. In fact, the thesis work benefitted from the early work of *Continual Queries*, a collaboration between Dr. Liu and Dr. Pu.

I benefitted much from group meetings and discussions with my fellow graduate students, especially the other three in the "Gang of Four": David Buttler, Wei Han, and Henrique Paques. We endured tough times together when we supported each other. The numerous BBQs and get-together parties were surely uplifting and colorful for the stressful life of a graduate student. I would also like to thank David Buttler for helping with the experiments in Chapter 3.

The entire CNS (Computing and Networking Services) group did their best in supporting

the college infrastructure for everyday computing needs. Special thanks are given to Dan Forsyth and Karen Carter, who were always responsive whenever I had a peculiar request for software or hardware setup. The administrative staff (Jennifer Chisholm, Deborah Mitchell, and Susie McClain) also helped to make organization and administration handling more smooth.

I am very grateful for the love and support from my family (my parents and my older sister). It was they who have always had belief in me and stood by my side. My parents taught me the value of education and worked hard to provide me the very best of it. My father inspired me to go abroad for higher education through the early endeavor of his own. It is my parents whose optimism has influenced me through my life.

My girlfriend - Weiling He, a PhD student studying architecture at Georgia Tech - has given me loving support during the past years. The world is much more colorful with her in my life.

The thesis work was supported primarily by grants from DARPA and NSF. I enjoyed their financial support, provided through my advisor, Dr. Ling Liu.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Information monitoring systems are publish-subscribe systems that continuously track information changes and notify users (or programs acting on behalf of humans) of relevant updates according to specified thresholds. Internet-scale information monitoring presents a number of new challenges. First, automated change detection is harder when sources are autonomous and updates are performed asynchronously. Second, information source heterogeneity makes the problem of modelling and representing changes harder than ever. Third, efficient and scalable mechanisms are needed to handle a large and growing number of users and thousands or even millions of monitoring triggers fired at multiple sources.

In this dissertation, we model users' monitoring requests using continual queries (CQs) and present a suite of efficient and scalable solutions to large scale information monitoring over structured or semi-structured data sources. A CQ is a standing query that monitors information sources for interesting events (triggers) and notifies users when new information changes meet specified thresholds. In this dissertation, we first present the system level facilities for building an Internet-scale continual query system, including the design and development of two operational CQ monitoring systems OpenCQ and WebCQ, the engineering issues involved, and our solutions. We then describe a number of research challenges that are specific to large-scale information monitoring and the techniques developed in the context of OpenCQ and WebCQ to address these challenges. Example issues include how to efficiently process large number of continual queries, what mechanisms are effective for building a scalable distributed trigger system that is capable of handling tens of thousands of triggers firing at hundreds of data sources, how to effectively disseminate fresh information to the right users at the right time. We have developed a suite of techniques to optimize the processing of continual queries, including an effective CQ grouping scheme, an auxiliary data structure to support group-based indexing of CQs, and a differential CQ

evaluation algorithm (DRA). The third contribution is the design of an experimental evaluation model and testbed to validate the solutions. We have engaged our evaluation using both measurements on real systems (OpenCQ/WebCQ) and simulation-based approach. To our knowledge, the research documented in this dissertation is to date the first one to present a focused study of research and engineering issues in building large-scale information monitoring systems using continual queries.

CHAPTER I

INTRODUCTION

The World Wide Web today has become a huge collection of information sources that changes continuously. As the Web grows and evolves, information change monitoring services are becoming increasingly useful. Instead of having users remember when to visit Web pages of interest and identify what and how the page of interest has been changed manually, the change monitoring service will deliver change information while it is still fresh. This kind of service is specially suited for busy individuals to keep track of transient data such as stock quotes, product prices, news headlines, and weather conditions. The monitoring requests are often modelled as "soft triggers".

Designers of large-scale information monitoring and change notification systems face a common problem: most software systems are based on a pure request-response model that does not allow servers to asynchronously notify clients of events on server side. From system perspective, many distributed systems need the functionality of asynchronous event dissemination. Examples include callbacks in distributed file systems [90] and gossip messages in lazy replication systems [57]. It is desirable to design a general-purpose event dissemination infrastructure on which large-scale change monitoring systems can be implemented.

1.1 Information Monitoring: issues and challenges

There are many technical challenges for building a scalable information monitoring system, especially in an open environment such as the Internet. We can only name a few of the most important ones in this section.

Source Heterogeneity:

Data can be classified as structured, semi-structured, or unstructured.

Structured Data has pre-defined schema or rigid data model. Examples of structured data include data residing in relational databases or object-oriented databases.

Semi-Structured Data is data whose structure is irregular or incomplete, does not have a fixed format, and can evolve. The schema of semi-structured data is usually descriptive rather than prescriptive and is often implicit in the data. Examples of semi-structured data include XML files and HTML pages (viewed in DOM model [75]). Data extraction tools and modelling languages (e.g., [1], [18], [8], [55], and [61]) are often used to manage semi-structured data or transform it to structured data.

Unstructured Data share similar characteristics as semi-structured data in that it also does not have a clear defined data model for efficient accessing, storing, and querying. We consider unstructured data to be the extreme of semi-structured data. Examples of unstructured data include some Web pages, books, emails, news articles, images, and PDF or Word documents.

Information on the Web does not share a global schema. The heterogeneous nature of the online data poses immediate challenges for information accessing, storing, and querying. In turn, it makes information monitoring more difficult than that in a closed corporate environment where data is often organized and centralized. New techniques are needed for accessing, storing, and querying the heterogeneous online data.

Event Detection and Change Tracking on the Net:
As the Web grows and evolves, we observe some rapid changes in the ways in which fresh information is delivered and disseminated. The mode of data transfer is shifting from a "pull-only" model to a "push-pull" model [2]. Many believe that the "push" style of information delivery and dissemination is, to some extent, a natural solution to the scale of the Internet. In the "push-pull" model, some data is *pushed* to users without an explicit pull request. The "push" style enables services to be served asynchronously as they become available. Instead of having users tracking when to visit web pages of interest and identifying what and how the page of interest has been changed manually, the information change monitoring service is becoming increasingly popular, which enables information to be delivered automatically while it is still fresh. The push service is specially suited for busy individuals and for delivering transient data such as stock quotes, product prices, news headlines, and weather information.

However, designers of large-scale Web-based change monitoring and notification systems face a common problem: HTTP [33] is a pure request-response model and it does not allow servers to asynchronously notify clients of events on the server-side. As a result, search engines to date, although powerful in helping users locating and finding information of interest, do not support tracking changes on behalf of users and cannot deliver timely information to the right users at the right time.

Now let us look at the problems from users' perspective, namely what is the common behavior of users who wish to monitor changes in web pages. Individuals often use a search engine to find a page of interest, and then bookmark the pages that they wish to re-visit. Upon a revisit, a fresh copy of the page will be obtained, and the user needs to determine if it has changed in an interesting way manually. Obviously the first challenge is the ability to allow users to only revisit a page *when* the page has changed in a way that is interesting. Furthermore, if a user becomes interested in tracking changes over a large number of web pages, he or she may wish to be notified not only when to re-visit the pages of interest but also what the concrete changes are. This is because when the number of pages of interest grows large it will be difficult, if not impossible, for a user to remember the concrete details about every page in which he or she was interested. Therefore, the second important challenge is to identify *what* and *how* the page has changed, including the types of changes and the amount of changes between the fresh copy and the copy last seen.

Up till now most tracking tool development [78, 104, 30, 108, 113, 21, 105, 106] has gone on at companies with little exposure of technical details, especially the efficiency, the scalability, and the tracking quality of such systems. Furthermore, from individual users' perspective, we observe three common problems with these tools. First, with the exception of Netmind [78] and AIDE [30], most of the tools only address the problem of when to re-visit a fresh copy of the pages of interest but not the problem of what and how the pages have changed. Second, all these tools handle the *when* problem with a limited set of capabilities. Furthermore, these tools treat each Web page tracking request as a unit of notification. Users who register a large number of pages with the tracking service are easily overwhelmed with the large number of frequent email notification messages.

Difference Generation and Presentation:

Most of the tools that monitor changes to web pages have the capability of notifying users that something on a page has changed with the link to the new copy of the page. But few are able to include what and how the page has changed in the notification report. When the number of pages that a user is interested in tracking changes is large and the changes on the pages are subtle, it is likely that the user will not know what has changed by simply viewing the new copy of the page and comparing it with what the user has remembered when the page was last seen. Therefore, computing and showing the difference to a web page is a critical component of an information monitoring system from the usability perspective.

One way to compute the difference between two versions of an HTML page is to use an HTML-aware difference utility (e.g., the HTMLDiff utility developed by [30, 29]). The new document highlights the differences between the two versions by flagging the inserted text with one coloring scheme and deleted text with another coloring scheme. Changes to existing text are treated as deletions followed by insertions. As pointed out by Fred Douglis and his colleagues [30], every invocation of the *HTMLdiff* may potentially consume significant computation and memory resources. Such resource overheads will restrain the number of difference operations a server can perform at a time, limiting the scalability of the system. In WebCQ, we have provided various types of page sentinels that are targeted at tracking changes to a page fragment rather than the entire page. A page fragment can be a specific HTML object (such as phrase, list, and table), an arbitrary text fragment, or a specific component of the page (such as links, images, and words). In such cases, a simplified difference generation algorithm should be used to reduce the overhead of the general *HTMLDiff*.

There are three popular ways to present the difference between two documents [30, 78, 76]. The first approach is to merge the two documents by summarizing all of the common, deleted, and inserted contents in one document, as is done in HTMLDiff [30]. The main advantage of this approach is that all the changes are embedded in one document with the common unchanged part of the two documents displayed only once. However, showing all the change information in one document may make it difficult to read especially when the

documents are large or there are many differences between the two.

The second approach for difference presentation is to display only the differences but omit the common parts of the two documents. The GNU Diff utility [37] falls in this category. This approach is beneficial for large documents or two documents with much in common. The drawback is that the change context is lost. For Web pages, this can also lead to confusing presentations.

The third approach is a *side-by-side presentation* of the differences between two documents. This method enables uses to view bi-directional changes to both the old and new documents. Although this method also has the problem of presenting difference for very large documents and documents with a lot of changes, the side-by-side presentation is most intuitive for visually comparing documents by humans.

Besides the three popular ways for presenting differences between documents, there are also approaches that present page changes through analysis of the pages' hyperlink structures. We can use a tree visualization to illustrate the relative changes. When in a closed enterprise environment, changes to other documents that have links within the target page often means relevant changes to the target page as well. The tree view works best when the page has little presentational change whereas the underlying links contained in the page changed. The tree visualization approach also works well for site-level change detection and among a set of relevant documents to depict the change relationships.

For a change monitoring system, we may need to consider various combinations of presentation techniques for information differences. It is often necessary to include both visual cues and quantitative measurement hints to the user about what has been changed and how the data is changed. This is not a trivial task, especially when facing different change semantics for various application domains.

Scalability:

A frequently asked question in Web information systems is the scalability of a service when the number of users and the number of monitoring requests grow into millions.

It is widely recognized that the scale can affect many components of the system: naming, authentication, authorization, accounting, communication and the use of remote resources.

Scale also affects reliability: when a system scales numerically, the likelihood that some host will be down increases; when a system scales geographically, the likelihood that some hosts fail to communicate with others increases. Scale can affect performance as well, in terms of system load and communication latency.

General solutions to scalability fall into four categories: replication, distribution, caching [79], and optimization. They can be both hardware-related and software-related.

Fortunately, when the number of users increases, common interests also grow. Consequently, *grouping common user requests* becomes a promising technique to exploit the similarity among user requests and group them together for more efficient new information detection and dissemination. Caching and optimization can also be employed to improve the system response time.

1.2 Continual Query Solution and Contributions

Continual Queries (CQ) [62, 64, 100, 68, 69] have been proposed as a general technique to monitor updates from a variety of sources on behalf of users. The detailed continual query semantics and specification are presented in Chapter 2.

When CQ systems detect an update of interest, the new information is pushed to the user through a notification service. We have developed two prototype systems during the course of the research study presented in this thesis. We have designed and implemented the *OpenCQ* system [62, 64, 100] for update monitoring for structured and semi-structured data sources (e.g., relational and object-oriented databases, wrapped Web sites). The *WebCQ* system is designed for tracking changes on arbitrary Web page [68, 69]. Both systems are our weapons to attacks of the problem of update monitoring in an open environment, such as the Internet.

The mediator/wrapper technology used by the Continual Query systems provides a good alternative solution for accessing and manage heterogeneous data. The continual query construct is a good solution for querying past, present, and future information.

In this dissertation, we first present the system level facilities for building an Internet-scale continual query system, including the design and development of two operational

CQ monitoring systems OpenCQ and WebCQ, the engineering issues involved, and our solutions. We then describe a number of research challenges that are specific to large-scale information monitoring and the techniques developed in the context of OpenCQ and WebCQ to address these challenges. Example issues include how to efficiently process large number of continual queries, what mechanisms are effective for building a scalable distributed trigger system that is capable of handling tens of thousands of triggers firing at hundreds of data sources, how to effectively disseminate fresh information to the right users at the right time. We have developed a suite of techniques to optimize the processing of continual queries and improve the system scalability, including an effective CQ grouping scheme, an auxiliary data structure to support group-based indexing of CQs, and a differential CQ evaluation algorithm. The third contribution is the design of an experimental evaluation model and testbed to validate the solutions. We have engaged our evaluation using both measurements on real systems (OpenCQ/WebCQ) and simulation-based approach. To our knowledge, the research documented in this dissertation is to date the first one to present a focused study of research and engineering issues in building large-scale information monitoring systems using continual queries.

1.3 Thesis Scope and Organization

This thesis presents the design and implementation of two prototype *Continual Query* systems that support push enabled data management and event-driven information delivery. The *OpenCQ* system is designed to handle change monitoring requests for structured and semi-structured data sources. The *WebCQ* system is aiming at providing update monitoring on the Web for arbitrary Web pages. Research challenges in change monitoring systems are addressed in both systems. A set of optimization techniques (namely continual query grouping and differential re-evaluation algorithm) are presented with the results demonstrating the effectiveness and efficiency of the proposed algorithms.

The thesis is organized as follows: Chapter 2 describes the *OpenCQ* system for structured and semi-structured information monitoring; Chapter 3 presents the *WebCQ* system

for change monitoring on arbitrary Web pages; Chapter 4 discusses the differential re-evaluation algorithm (DRA) for optimizing continual query evaluations; Chapter 5 explores ways to facilitate business information exchanges using change monitoring systems (such as WebCQ); Chapter 6 presents the related work; Chapter 7 concludes the dissertation and discusses future work.

CHAPTER II

STRUCTURED AND SEMI-STRUCTURED INFORMATION MONITORING

We usually refer to data with pre-defined schema or rigid data model as *Structured Data*. Examples of structured data include data residing in relational databases or object-oriented databases.

Semi-Structured Data is data whose structure is irregular or incomplete, does not have a fixed format, and can evolve. The schema of semi-structured data is usually descriptive rather than prescriptive and is often implicit in the data. Examples of semi-structured data include XML files and HTML pages (viewed in DOM model [75]). Data extraction tools and modelling languages (e.g., [1], [18], [8], [55], and [61]) are often used to manage semi-structured data or transform it to structured data.

Unstructured Data share similar characteristics as semi-structured data in that it also does not have a clear defined data model for efficient accessing, storing, and querying. We consider unstructured data to be the extreme of semi-structured data. Examples of unstructured data include some Web pages, books, emails, news articles, images, and PDF or Word documents.

In this chapter, we study the problems of information monitoring for *Structured* and *Semi-Structured* data. We present our solution of using *Continual Queries* to monitor data changes. We use mediator/wrapper technology and a set of toolkits [61, 118] to transform semi-structured data into structured data sources.

2.1 Overview

Continual queries are standing queries that monitor updates and return results whenever the updates have reached specified thresholds. A continual query consists of three key components: query, trigger, and stop condition. In contrast to ad-hoc queries in conventional

DBMSs or web search engines or query systems, a continual query, once issued, runs continually over the set of information sources, until its stop condition is satisfied. Whenever its trigger condition becomes true, the new result since the previous execution of the query will be returned. The trigger part of a continual query specifies events or situations to be monitored. We distinguish primitive events from conditional (logical) events and allow events to be composed of other events. We use primitive events to model basic database operations (such as INSERT, DELETE, UPDATE), basic time events (such as at time-specification, every time-period, and after time-period), or signals from arbitrary processes. We use conditional events to model various conditional situations to be monitored. We provide a rich set of event composition operators (such as logic operators: conjunction, disjunction, negation; and execution dependency operators: serial, serial alternative, parallel, parallel alternative) to support composition of events.

Continual queries are useful both to external applications and as a convenient mechanism for implementing push-based data delivery functions beyond conventional storage, retrieval, and update of data in conventional DBMSs.

2.1.1 CQ Concepts and Semantics

A continual query is defined by a quintuple (Q, T_{cq}, `Start`, `Stop`, `Notification`), consisting of a normal query Q (e.g., an SQL-like query, an XQuery, or a keyword-based query), a trigger condition T_{cq}, a start condition `Start`, a termination condition `Stop`, and a notification condition `Notification`. When `Start` condition is omitted, the continual query assumes to start at the time of installation. When `Notification` condition is omitted, the CQ is assumed to use the default notification settings [1]. In contrast to *ad hoc* queries in conventional DBMSs or Web search engine queries, a continual query, once activated (i.e., installed and started), runs continuously over the set of information sources. Whenever the trigger condition T_{cq} becomes true, the query will be fired. If the user specifies a differential result from the last execution, upon the detection of a difference between the current query result and previous query result, a notification will be sent to the user in the form of an

[1] You will see continual queries expressed as triplets of (Q, T_{cq}, `Stop`), which is also legit.

email[2]. Otherwise, the notification will include the current query result.

Continual Semantics: Let us denote the result of running query Q on data source state S_i as $Q(S_i)$. We define the result of running a continual query CQ as a sequence of query answers $\{Q(S_1), Q(S_2), \ldots, Q(S_n)\}$ obtained by running query Q on the sequence of data source states $S_i (1 \leq i \leq n)$. $Q(S_1)$ starts when the `Start` condition is true. The subsequent execution of $Q(S_i)$, at each given state S_i $(i > 0)$, is triggered by $T_{cq} \wedge \neg$`Stop`.

The initial execution of a continual query is performed as soon as its Start condition is verified. The first run of its query component Q is performed over past and present data represented by the state of information sources, and the integrated result obtained by executing Q is returned to the user. The subsequent executions of Q are performed whenever a new update event is signaled and the trigger condition T_{cq} becomes true. For each subsequent execution of Q, only the new query matches since the previous execution are returned to the user unless specified otherwise. Thus the domain of continual queries is defined over past, present, and future data, whereas the domain of pull queries is limited to past and present data.

We support two types of events: *time events*, which involve clock times, dates, and time intervals; and *object events*, which involve changes to non-temporal objects. Accordingly, we distinguish three types of trigger conditions: time-based trigger condition, which consists of only time events; content-based trigger condition, which consists of only object events; and the hybrid trigger condition, which consists of any combinations of time events and object events. Three types of temporal events are supported for time-based or hybrid trigger condition:

- absolute points in time, defined by the system clock (e.g., 14:30:00, November 25, 2003);
- regular time interval (e.g., every Monday or every two weeks) or irregular time interval (e.g., every first day of the month);

[2]Whether a user receives an immediate notification or a delayed one also depends on the notification condition for the CQ subscription. For example, a user may want to receive CQ reports every two weeks.

- relative temporal event (e.g., 50 seconds after event A occurred).

A *content-based* trigger condition can be defined in terms of a database query, a built-in situation assessment function (e.g., the inventory level of any item changes), a user- defined method (e.g., the price of IBM stock drops by 5%), or an application-generated signal (e.g., an abnormal signal from a diagnostic routine on a sensor at the water temperature sampling station). Furthermore, the trigger conditions to be monitored may be complex and can be defined:

- on sets of distributed data objects (e.g., the total of pending legal cases exceeds the given threshold),
- on transitions between states (e.g., the new position of the ship is closer to the destination than the old position),
- on trends and historical data (e.g., the output of the sensor increased monotonically over the last two hours), or
- on a relationship between a previous query result and the current database state (e.g., the water temperature at Tansy Pt. changes 20% since the last reporting time).

Furthermore, both the `Start` condition and the `Stop` condition can be specified in terms of time events or object events. Both the trigger condition T_{cq} and the termination condition `Stop` are evaluated prior to each subsequent execution of the query component Q. In the OpenCQ prototype, we restrict the `Start` and `Stop` conditions to be time events to simplify the implementation effort, since most frequently used Start and Stop conditions are time events. By default, we use content-based (or object-based) trigger conditions since they specify the real users' interests. The complete BNF description of the CQ event specification and the formal semantics of continual query specification model are documented in Appendix A. We demonstrate the continual query concept in the following example:

Example 1 *Consider continual query "Notify me (john.doe@blah.com) whenever the current price of Microsoft stock drops by 5% in the next 8 months". It can be expressed as follows:*

```
Create CQ Microsoft_stock_watch as
Query:   EXTRACT daylow, dayhigh, volume
         FROM    stock@stockmaster.com
         CONDITION  symbol = 'MSFT';
Trigger: AT SOURCE stock@stockmaster.com
         WHEN stock.price DECBYP 5
         WHERE stock.symbol = 'MSFT';
Notif:   email(john.doe@blah.com);
Start:   9:00:00 EST, Oct. 20, 2003;
Stop:    17:00:00 EST, Dec. 19, 2003;
Calendar: Monday through Friday, 8:00:00 am
          through 5:00:00 pm;
```

There are two types of continual queries in terms of the correlation between the trigger and query components. For the *simple case*, a user may be only interested in being notified when source changes happen. In this case, the query component is hidden. For the *complex case*, a user may specify complicated query components in react to the triggered events. The query may be set on a set of data sources different from the triggering sources. Again, a user may request a *punctual query* result that returns the current result when the query is evaluated; or a *differential query* result that returns the diff'ed content between the current and previous query results.

For CQ notifications, a natural mechanism is through email, in which either the query result content or a link to the content can be returned to the user. In current CQ system, we adopt the latter approach in consideration of reducing traffic caused by massive email notifications. Thus, the notification result is only shipped to the user when he/she explicitly requests it.

2.1.2 System Architecture of OpenCQ*

OpenCQ* is a continual query system based on previous OpenCQ framework [62] with built-in multi-level grouping algorithms[3]. The system architecture is shown in Figure 1.

[3]Except for this difference, the main components of OpenCQ* and OpenCQ are the same. For the rest of the thesis, we still refer to OpenCQ* as OpenCQ.

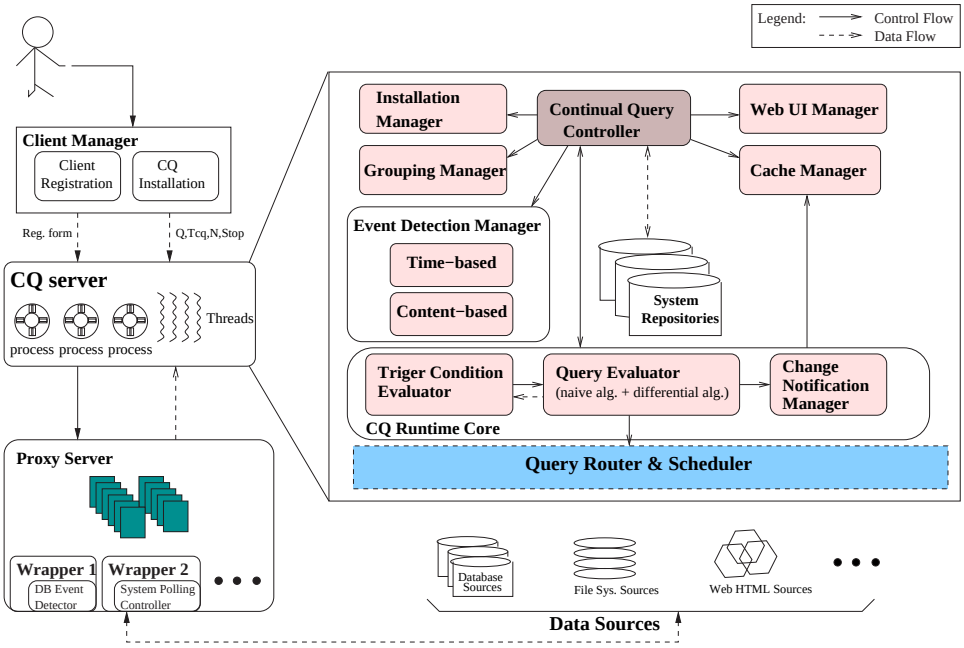

Figure 1: OpenCQ* System Architecture

At the time a continual query is installed, a persistent CQ object will be created. A change detection query and a trigger evaluation expression will be derived from the trigger component. The CQ is classified into one of the processing groups according to its trigger expression (explained in details in Section 2.4). The CQ runtime daemon is in charge of monitoring information source changes and checking of user-defined thresholds to react to these changes.

The main idea behind grouping based on trigger expressions is that many of the expressions share the same or similar structures. For example, a lot of people may be monitoring the same information (same CQ). Or they would want to watch the same information source, but have different reactions to the changes (same trigger, different queries). Users could also have overlapping interests (partial match for trigger expression), e.g., a user might be interested in IBM and Microsoft's stock price changes while another is interested in IBM and Nokia's stock prices.

Query caching is employed in CQ system since query results can be reused for multiple users who may query the same data source in response to changes found by same or different triggers.

Email notifications are grouped for each user because a single summarized email to the user is for his/her convenience and a relief for the system load.

A *Continual Query* has three main components (query, trigger, and notification) for its execution. Previous research [25, 43, 7] focuses on global optimization in trigger evaluation only. We argue that in order to achieve maximum scalability of a continual query system, we have to exploit parallelism and optimization in all three levels in CQ execution, namely trigger grouping, query caching, and notification grouping.

In the following sections, we study the different grouping techniques to exploit concurrent processing opportunities at trigger level, query level, and notification level. We focus on the trigger level grouping (since it is the most important step in CQ processing) and demonstrate the benefit and trade-off of grouping, as well as the general runtime characteristics of continual query processing.

2.2 Continual Query Execution Semantics

We have explained how one defines continual queries in the previous section. We now describe the implementation of how the OpenCQ system triggers and executes continual queries.

It is well known that in a conventional pull-based DBMS user application programs are executed when explicitly requested to do so. Execution of such programs typically results in the processing of a sequence of *transactions*, where each transaction is a unit of consistency and recovery. The system guarantees *atomicity* (all updates issued by the transaction are installed in the database or none are), *serializability* (the concurrent interleaved execution of a set of transactions is equivalent to a serial no-interleaved execution), and *durability* (once a transaction is committed, its updates will never be rolled back). In contrast, a continual query system must evaluate installed continual queries under system control (not user or application control). More concretely, once a continual query is installed, the system must decide how to detect the update events of interest, how to evaluate the trigger condition, and when to fire the subsequent execution of the query component, and how should the execution of these tasks be treated with respect to user transactions? This section is an

attempt to answer these questions.

2.2.1 Basic Coupling Modes

Continual queries in practice are often defined over multiple, autonomous and possibly heterogeneous data sources. The local update transactions are usually orthogonal to the continual queries specified over the same set (or a subset) of data. Furthermore, both trigger condition evaluation component and query component of a continual query are side-effect free transactions. Due to the autonomy and distribution of data sources and the side-effect free nature of continual queries, it is not only important but also practical to allow a more flexible execution model.

A flexible execution model allows trigger condition evaluation and query execution to be broken off into different execution threads from the triggering transaction (the transaction that carried out the update operations). More concretely, it should be possible to allow the continual query evaluation to be separated from the (triggering) transaction that carried out the actual updates. This would allow the triggering transaction to commit earlier, and would potentially increase concurrency and reduce wasted work (rollback of incomplete transactions after a crash). The OpenCQ execution model for continual queries uses the notion of coupling modes to provide this flexibility.

In OpenCQ we support four basic coupling modes: transaction coupling mode: *separate* or *same*, execution coupling mode: *asynchronous* or *synchronous*, dependency coupling mode: *causally dependent* or *causally independent*, and schedule coupling mode: *immediate* or *deferred*. We view the execution model of each continual query to consist of the following four participating transactions:

- (1) the triggering transaction that carries out the update operations,
- (2) the update event detection transaction that detects if the data of interest has been updated,
- (3) the trigger condition evaluation transaction that evaluates the condition based on the newly updated data, and

- (4) the transaction that carries out the subsequent execution of the query component and sends out the alerts or change notification messages.

Such arrangement provides more flexibility for utilizing multiple execution threads and parallel execution for continual query processing, which are critical techniques to the effectiveness and responsiveness of an event-driven distributed information delivery system.

In OpenCQ, it is possible that the coupling case for transaction types (1) and (2) may be different from the coupling case for transaction types (2) and (3) as well as the coupling case for transaction types (3) and (4).

We illustrate the meanings of each coupling mode using the coupling scenario for transaction types (2) and (3), which relates to the trigger condition part of the continual queries. For the trigger condition part of a continual query, the coupling mode specifies when the condition is to be evaluated relative to the triggering event (i.e., the update event being monitored):

- **Transaction coupling mode**: separate or same

  The transaction coupling mode *separate* means that the condition evaluation triggered by the update event runs as a separate transaction with respect to the transaction that detects the update events of interest.

  The transaction coupling mode *same* means that the condition evaluation triggered by the update event runs either as part of the transaction for detecting the update event in the case that the updates performed by the triggering transaction are local operations, or as part of the triggering transaction in the case that the updates are performed by the same user or application program who installed the continual query.

- **Execution coupling mode**: asynchronous or synchronous

  The asynchronous coupling mode means that the update event detection transaction may run in parallel with the trigger condition evaluation transaction.

  The *synchronous* coupling model means that if the trigger condition evaluation transaction is triggered by the transaction that detected the update events, then the trigger

condition evaluation transaction is executed, and the execution control returns to the 'triggering' transaction only after the condition evaluation transaction is committed.

- **Dependency Coupling Mode**: casually dependent or casually independent

  The *casually dependent* coupling mode means that the trigger condition evaluation transaction can be scheduled only after the 'triggering' transaction that detected the update events has committed.

  The *casually independent* coupling mode means that the scheduler is free to schedule the trigger condition evaluation transaction independently of the update event detection transaction when the update transaction is local.

- **Schedule Coupling Mode**: immediate or deferred

  The schedule coupling mode *immediate* means that the trigger condition evaluation transaction is fired as soon as the triggering transaction commits. When the updates are carried out by a global update transaction issued by the same user or application program, the triggering transaction refers to this global update transaction. When the updates are carried out by local transactions or other remote and autonomous transactions, the triggering transaction refers to the update event detection transaction.

  By looking into the semantics implication of these coupling modes, We come to the following conclusion: The schedule coupling mode *deferred* must be used in conjunction with the *same* transaction coupling mode. This mode means that the CQ trigger condition evaluation is fired at the end of the update event detection transaction and before it commits.

The *same* transaction coupling mode can be used only in conjunction with synchronous execution coupling. The *separate* transaction coupling mode can be used only with *asynchronous* execution coupling. The *deferred* schedule mode is applicable only in conjunction with the *same* transaction coupling mode. However, the immediate schedule mode can be

used in conjunction with both *same* and *separate* transaction couplings. Also both dependency couplings are applicable only to *separate* transaction coupling, *immediate* schedule coupling, and *asynchronous* execution coupling.

In a similar manner, we may illustrate the possible coupling cases for transaction types (1) and (2), the event detection part of the CQ, and for transaction types (3) and (4), the query scheduling part of the CQ. For the query scheduling part of a CQ, each coupling case specifies when the subsequent run of the query component is to be fired relative to the trigger condition evaluation transaction.

In OpenCQ we allow users to define their application-specific coupling modes for any of the three pairs of the participating transaction types. In the absence of user-specified coupling modes, the system default coupling case will be used. They are separate, asynchronous, causally independent, and immediate for all the three coupling scenarios.

2.2.2 Continual Query Installation

Once a continual query CQ_i, denoted by $(Q, T_{cq}, \texttt{Stop})$, is defined, the user may install it directly to the OpenCQ continual query system. At the installation time, the `Install` module of the client manager takes the continual query and passes it to the OpenCQ continual query server. The server activates it using the `activate` command. The activation process consists of the following three main tasks:

- making this continual query a persistent object and generating a unique identifier (`cqid`) for it;
- Modifying the expression of the query component to incorporating the trigger condition semantics. This task is accomplished by checking if the trigger condition component T_{cq} and the query component Q are defined over the same set of data, i.e., DataSet(Q) = DataSet(T_{cq}), where DataSet(Q) is the set of instance variables used in Q and DataSet(T_{cq}) is the set of instance variables used in T_{cq};
  - if yes, merge the trigger condition into the `WHERE` clause of the query component Q, and denote the modified query expression as Q', execute Q' instead of Q for

the first run of CQ_i, and cache the answer as the previous run result;

– if not, identify if there is a common part of the data set shared by T_{cq} and Q, i.e., checking if DataSet(Q) $\cap$ DataSet(IT_{cq}) $\neq \emptyset$, if yes, merge the common portion of the T_{cq} into the query component Q, and denote the modified query component by Q'; otherwise, let $Q' := Q$; then perform the following two actions: (1) execute Q' for the first run of CQ_i and cache the answer as the previous run result of the query component; (2) fetch the other portion of T_{cq}, i.e., DataSet(T_{cq}) − DataSet(Q), and cache the result for the subsequent trigger condition evaluation of the CQ_i;

- Initializing the execution attributes and data structures used for event detection and condition evaluation of this given CQ. This task includes decomposing the user-specified CQ trigger condition into a set of triplets, each triple is described by a basic update event, an atomic conditional event, and a connector; and setting up the initialization for the transaction coupling mode, the dependency coupling mode, the schedule coupling mode, and the execution coupling mode (recall Section 2.2.1).

The `Activate` command also returns a handle that will be used to deactivate this continual query when its termination condition is expired.

Users can use the `activate` command to define the coupling modes according to application specific requirements. The syntax of the `activate` command is given below:

```
Activate <cqid>
  define communication protocol between
      <trans1> and <trans2>
  TransactionCoupling = same | separate
  ExecutionCoupling = synchronous | asynchronous
  DependencyCoupling = causally dependent | causally independent
  ScheduleCoupling = immediate | deferred
```

Once a continual query is activated, it runs continually following the communication protocol defined by the specific coupling case. The continual query is terminated when its `Stop` condition is evaluated to be true. To terminate an installed continual query, the

command `Deactivate <cqid>` is invoked, which removes from the OpenCQ system catalog the corresponding continual query object identified by `cqid`, deactivates the related event detectors that are still active, and sends to the owner of this CQ a notification that this CQ is expired.

2.2.3 Event Detection

The main task of event detection manager is to decide what to detect, when to detect, and how to detect. The decision is made based on the update events identified from the trigger condition specification and the type of events to be detected. The trigger condition part of a continual query may be a primitive event, such as a *temporal event*: every two days or every first day of the month; an *atomic conditional event*: the stock price is greater than 100 (`price > 100`); or a *composite event*, which is formed by an event composition expression of the form "E_1 `<event_op>` E_2", where E_1 and E_2 are primitive or composite events. Typical examples of composite events are

```
Stock.price('IBM') IncreaseBy% 5 OR Stock.price(Intel) DescreaseBy% 5
keyword CONTAINS 'Java' OR keyword CONTAINS 'JDBC'
qty_on_hand(item) > threshold(item)
qty_on_hand(item) + qty_on_order > threshold(item)
```

Each primitive event is detected by using a primitive event detector, which is either a basic temporal event detector or an atomic conditional event detector. An operation *signal* is defined for the event entity type, and is executed by the event detector components of the system.

2.2.3.1 Time-based Event Detection

For time-based continual queries, a temporal event detector, or so-called time-based event detector, is used, which translates the time-based trigger condition into a clock event and installs the clock event script to the OpenCQ clock daemon. Whenever the clock event occurs, the trigger condition is signaled. Thus the subsequent execution of the query component is fired. A distinct feature of time-based continual queries is the use of *user-controlled polling* for update monitoring.

There are two key implementation techniques useful for time-based event detection:

The first technique is to design a generic transformation program that takes the user-defined time condition and transforms it into a clock event expressed in the clock event scripting language; the clock manager (daemon) will then take over the control and trigger the update event detection according to the clock event installed; whenever the update event is signaled, the continual query manager will call the query evaluator to fire the subsequent run of the query component, and call the change notification manager to deliver the change notification message as well as the update result. The second technique is to develop a clock event manager which, on one hand, provides a scripting language to allow users to specify an arbitrary clock event and the action to be taken if the clock event occurs, and on the other hand, provides triggering capability so that it can fire the specified action (e.g., invoke a program) when a specific clock event is signaled.

The implementation of a clock manager is a system-specific decision. One may either choose to design a clock manager specifically for this purpose, or reuse the clock manager provided by an operating system (such as Cron by Unix and Scheduler by NT). In the first prototype of the OpenCQ system, we make use of Cron as the clock manager. We are considering to write our own clock manager in the next prototype release to further enhance the efficiency of the system.

2.2.3.2 Content-based Event Detection

In contrast to time-based continual queries, the content-based continual queries use the *system-controlled polling* for update monitoring. Thus, there are more than one strategies possible for implementation of the CQ trigger condition monitoring and event detection.

In order to carry out the content-based event detection, the first thing we need to do is to identify what update events are of interest to the given continual query. As mentioned in the continual query activation procedure (recall Section 2.2.2), for each installed continual query (Q, T_{cq}, `Stop`), its trigger condition T_{cq} is decomposed into a list of T_{cq} triplets, each triple is described by a basic update event, an atomic conditional event, and a connector. For example, if the trigger condition is "`Stock.price(IBM) IncreaseBy% 5 OR Stock.price(Intel) DescreaseBy% 5`", then the following triplets are generated:

```
(Stock.price, Stock.price IncreaseBy% 5, WHERE)
(Stock.company, Stock.company = IBM, OR)
```

```
(Stock.price, Stock.price IncreaseBy% 5, WHERE)
(Stock.company, Stock.company = Intel, END)
```

For the trigger condition: `qty_on_hand(item) > threshold(item)`, two triplets are generated. They are: `(qty_on_hand, true, >)` and `(threshold, true, END)`. Note that the connector `WHERE` means that the next triple is not an update event of interest but a constraint on the current update event. In this case, UPDATE on the stock price is the event we would like to monitor, and the condition `Stock.company = IBM` is simply a constraint, saying that we are only interested in monitoring UPDATE on the stock price of IBM but not other companies' stock prices.

Now we can determine **what to detect** based on the basic update events identified by the list of T_{cq} triplets.

Example 2 Given the trigger condition:

"`Stock.price(IBM) IncreaseBy% 5 OR Stock.price(Intel) DescreaseBy% 5`",

the basic events of interests are UPDATE operations on `Stock.price` and `Stock.company`, as well as INSERT and DELETE operations on the object class `Stock`. For trigger condition `keyword CONTAINS 'Java' OR keyword CONTAINS 'JDBC'`, if the condition field name `keyword` is mapped to `Document.title` and `Document.abstract` available at the corresponding data source(s), then the basic events of interests are INSERT and DELETE operations on `Documents` objects, and UPDATE operations on `Document.title` and `Document.abstract`.

The next question is **how to detect**, namely we need to decide which mechanisms may be used to detect the changes made by the update operations, possibly from some transactions that are local to the data source; In OpenCQ, we distinguish between the data sources that have built-in trigger capability such as the data sources managed by trigger-enabled RDBMSs (including Oracle, DB2, Informix, Sybase) and the data sources that have no built-in trigger capability such as most of the web sites and file systems.

- For the data sources with built-in trigger facility, the OpenCQ system may install the database triggers on the data columns or objects of interest. Whenever there is an update, the database transaction that carries out this update will send an update signal to the corresponding CQ wrapper. We provide the host-specific trigger installation program (such as Oracle trigger installation program) to install triggers on those data objects and data columns that are accessible to the OpenCQ continual query system.

- For the data sources with no built-in trigger facility, we use system-controlled polling with system-defined interval (such as every 30 seconds).

Note that the capabilities of database trigger supported in commercial DBMSs today are not sufficient, particularly in those cases where run-time installation of customized database triggers is required, In these situations, a system-controlled polling will be used in conjunction with the database triggers. Our experience tells that not all the RDBMSs allow database triggers to be installed by a remote program through JDBC. In the first prototype of OpenCQ, we implement the content-based event detection using the system-controlled periodic polling.

Now, let us walk through the event detection process. Given a continual query CQ_i defined by $(Q, T_{cq}, \texttt{Stop})$. Suppose that the trigger condition T_{cq} has been transformed into a list of T_{cq} triplets, denoted by TripleSet($\texttt{cqid}, T_{cq}$). To simplify the steps (that) we need to walk through, let us assume that the connectors we use in this walkthrough are the most commonly used ones, namely `WHERE, AND, OR, END`. For each triplet in TripletSet($\texttt{cqid}, T_{cq}$), we form a event detection query, denoted by Q_{detect}, which is to be submitted to the relevant data sources to detect if an update is occurred.

- For a triple of the form $(T.A, T.A\vartheta v, \texttt{AND})$ or $(T.A, T.A\vartheta v, \texttt{OR})$ or $(T.A, T.A\vartheta v, \texttt{END})$, where T denotes the object class, A, B are instance variables of T, and ϑ is the comparison operator, let `prev` denote the value of instance variable A contained in the result of previous execution of the given CQ. Thus, the corresponding event detection query Q_{detect} is expressed as `SELECT A FROM T where A` $\neq$ `prev`.

- For a triplet of the form $(T.A, T.A\vartheta w, \texttt{WHERE})$, we fetch the next triple, say

$(S.B, S.B\vartheta w, \texttt{END})$ from the remaining list of TripletSet($\texttt{cqid}, T_{cq}$). Thus, the event detection query Q_{detect} is expressed as `SELECT T.A, S.B FROM T, S WHERE S.B` ϑ `w AND T.A` $\neq$ `prev`.

2.2.4 Condition Evaluation

In principle, one may want to detect all the update events of interest before starting the trigger condition evaluation process. In practice, the CQ trigger condition evaluation is carried out in conjunction with the process of basic update event detection, to guarantee the efficiency of the condition evaluation. For example, if a condition is of the form $(T.A\vartheta v_A) \wedge (T.B\vartheta v_B)$, and if the event detection query over the triplet $(T.A, T.A.\vartheta v_A, \texttt{AND})$ returns empty answer, then we can conclude that the trigger condition is false without looking into the second triplet $(T.B, T.B\vartheta v_B, \texttt{END})$.

Now, let us walk through the condition evaluation process for a continual query CQ_i defined by $(Q, T_{cq}, \texttt{Stop})$. Let TripleSet($\texttt{cqid}, T_{cq}$) denotes the list of T_{cq} triplets generated by the CQ activation process. Similar to the discussion on event detection, we simplify the steps we need to walk through by assuming that the connectors used in this walkthrough are `WHERE, AND, OR, END`. The condition evaluation process of CQ_i proceeds as follows:

- Step 1: It starts by selecting a triple in TripleSet($\texttt{cqid}, T_{cq}$), and then check the connector type of this triple:
- Step 2: if it is an `END` connector, then this content-based trigger condition is evaluated to be true, and the subsequent query execution is fired.
- Step 3: if it is an `WHERE` connector, let us denote the selected triple as $(T.A, T.A\vartheta v, \texttt{WHERE})$, and the next triplet is fetched from the remaining list of TripleSet ($\texttt{cqid}, T_{cq}$), denoted by $(S.B, S.B\vartheta w, \texttt{AND})$, then the update event detection query Q_{detect} is expressed as `SELECT T.A, S.B FROM T, S WHERE S.B` ϑ `w AND T.A` $\neq$ `prev`. If Q_{detect} returns a non-empty answer, it means the update event has occurred; go to step 6. If Q_{detect} returns an empty answer, we can conclude that the corresponding trigger condition is false.

- Step 4: if it is an `AND` connector, let us denote the selected triple as $(T.A, T.A\vartheta v, \texttt{AND})$, then the update event detection query Q_{detect} is expressed by `SELECT T.A FROM T where T.A` $\neq$ `prev`. If the answer to this query Q_{detect} is empty, then the condition evaluation is false. Otherwise (i.e., if the answer is non-empty), go to Step 6.

- Step 5: if it is an `OR` connector, let us denote the selected triple as $(T.A, T.A\vartheta v, \texttt{OR})$, then the update event detection query Q_{detect} is the same as the case for an `AND` connector, i.e., `SELECT T.A FROM T where T.A` $\neq$ `prev`. However, unlike the `AND` connector case, if the answer to this query Q_{detect} is non-empty, then we conclude that the condition evaluation is true. Otherwise (i.e., the answer is empty), we need to go to Step 6.

- Step 6: select another triplet from the remaining list of triplets in TripleSet($\texttt{cqid}, T_{cq}$), and go back to Step 2.

Obviously, the richer set of event composition operators is used, the more sophisticated the event detection process will be. A complete description of event composition operators and their formal semantics is beyond the scope of this paper.

2.2.5 Issues on Efficient Condition Evaluation

Users and application programs may define as many continual queries as they wish. Once these continual queries are installed, they run continually as long-running side-effect free transactions with checkpoints [4]. Despite all the query components, each from one installed continual query, the set of all trigger conditions forms a potentially large set of predefined queries (i.e., event detection queries) that have to be evaluated efficiently. Furthermore, the trigger condition component of a continual query may be more sophisticated than the query component when the update monitoring threshold is defined over several different object classes and uses special operators (such as `IncreaseBy%`) that are not supported by the data sources upon which the condition is evaluated. Several techniques have been identified as being useful for performance optimization of the condition evaluation:

[4] Each time when the trigger condition is evaluated to be true and the query is fired is referred to as a checkpoint.

The first technique is *Multiple Condition Optimization* and also called multiple query optimization in the literature [94]. This technique represents conditions (and the events that signal the condition evaluation) by condition evaluation graphs, which resemble the query graphs commonly used in query processing. The leave nodes of the graph are triples of the form $(R, R+, R-)$, where R corresponds to a set of entity instances before the update, $R+$ corresponds to the set of instances inserted into R by the update, and $R-$ the set of instances deleted from R by the update. The internal nodes correspond to operators of some convenient algebra into which the query language can be compiled (e.g., select, project, join). The key idea of multiple condition evaluation consists of identifying common subgraphs, and evaluating these subconditions once for a whole set of queries, instead of once for every query [87, 94]. For a continual query system, the common subconditions may be detected at the algebraic level due to the distribution and autonomy of data sources, whereas in a centralize data base system the common subconditions may also be detected at the lower level (e.g., use common access paths). The multiple query evaluation problem is complicated by the need to ensure that the conditions will have to be evaluated simultaneously; e.g., they are triggered by the same update event.

The second technique is *Incremental Condition Evaluation.* A main task of continual query evaluation is to determine whether the answer to a previous execution of the query component (say at time t) has changed as a result of some update event to some of the query's operands at time t'. Let Q be a query defined over an entity set R, and $Ans(Q, t)$ be the answer to the query Q at time t. Let $R' = (R \textit{ minus } R- \textit{ union } R+)$. A brute force method for computing the change in $Q(R, t)$ would be to compute $Ans(Q, t') = Q(R')$, and then the symmetric difference of $Ans(Q, t)$ and $Ans(Q, t')$. Incremental evaluation computes this symmetric difference directly from $R+, R-$, and Q. Sometimes R is also needed when Q involves joins [60]. Many algorithms have been proposed in view materialization research for incremental maintenance of materialized views (see Section 6 for reference), and may be directly deployable for incremental condition evaluation in the continual query systems.

An extreme case of incremental condition evaluation is the situation where it may be possible to infer that there is no change in a query's answer with respect to an update

event without evaluating the query. Put differently, we can ignore an update event E at t' with respect to the execution of query Q at t, if we can tell that the symmetric difference between $Ans(Q,t)$ and $Ans(Q,t')$ is empty by looking only at the update event E and query expression Q. A trivial example is the update event that modifies a data object that is irrelevant to the query Q. A less trivial example is an update that modifies the Intel stock price to a higher value; clearly, this update event is ignorable with respect to the trigger condition `Stock.price(Intel) DecreaseBy% 5`.

Also more opportunities for optimization may arise out of the interplay between the event detection, the condition evaluation, and the subsequent execution of the query component. Generally speaking, more work is needed to develop heuristics and cost models that the condition monitor can use to explore the tradeoffs and benefits of these tactics and algorithms.

2.2.6 Parallel Processing of Continual Queries

It is commonly recognized that using parallelism and concurrency of multiple tasks to manage CPU and I/O resources can give better throughput and response time even for a single processor. In OpenCQ we promote the use of parallel processing as an important way to obtain better scalability. A number of parallel processing methods are currently exploited for improving scalability of OpenCQ in processing large numbers of continual queries.

In the previous sections we discussed concurrent processing strategies used in the OpenCQ continual query processor at query level, trigger level, event level, and data level. The *query-level* of parallelism refers to the strategy that processes a CQ query at multiple web information sources concurrently. The *event-level* of parallelism is also called trigger-pattern-level of concurrency, by which we mean that multiple groups of CQ triggers can be processed in parallel through the use of CQ indexes on trigger patterns. Each trigger pattern is processed at the set of information sources selected by the query router through the event observers at the OpenCQ wrapper tier (recall Section 2.1.2). Thus each trigger pattern corresponds to multiple concurrent change event observation tasks. The *trigger-level* of concurrency

means that multiple trigger conditions within each partition group can be tested concurrently against a single set of data objects. These objects are often fetched remotely using the indexed trigger pattern and filtered at the wrapper tier and the mediator tier for each CQ conforming to the given trigger pattern. The *data-level* of concurrency refers to the support of concurrent access to the same set of data objects.

In order to capitalize on the various types of concurrent processing, OpenCQ has been implemented primarily using Java multi-thread programming language features. Instead of using a task queue kept in shared memory to store incoming or internally generated work, the current implementation of the OpenCQ system spawns multiple threads to execute multiple tasks rather than explicitly managing a task queue. Once a continual query is fired, a CQ driver process is invoked. Multiple driver processes can call `OpenCQ` trigger evaluator concurrently, each driver process makes such a call every T time units (such as every 200 milliseconds for local processes and every 5 minute for remote connection). The default value of T should allow timely trigger execution without excessive or unnecessary overhead for communication between the driver processes and the remote information source servers. The factors to be used to determine the best value of T is an interesting issue in our list of further work. Once a driver program is completed, it waits for T time units to make the next call to `OpenCQ` again.

Another implementation goal is to keep the execution time inside `OpenCQ` reasonably short. A long execution could result in higher probability of faults such as running out of memory or deadlocks and the problem of excessive work to be lost if a rollback occurs during continual query processing. The details of continual query recoverability feature are beyond the scope of this paper and interested users may refer to our upcoming technical reports.

2.3 Event Observers for Semi-Structured Data on the Web

Most web sources either have no built-in trigger capability or do not export such change notification function, which can locally detect and notify a change when it occurs. As a consequence, `OpenCQ` needs to build a change event observer for monitoring and detecting

changes at a remote web information source. Given the variety of web data sources, this requires careful thought.

A change event observer is a program that performs a pull query or pull-based page fetch request periodically over a remote information source and discovers what changes were occurred recently and what are the types of these changes. Currently, we are designing and developing content-based change event observers that can detect the specific content changes as well as the types of content changes. All `OpenCQ` content-based event observers are built on top of the wrappers generated semi-automatically using the XWRAP technology [61]. The wrappers are responsible for accessing and transforming the semi-structured data on the Web to structured data sources.

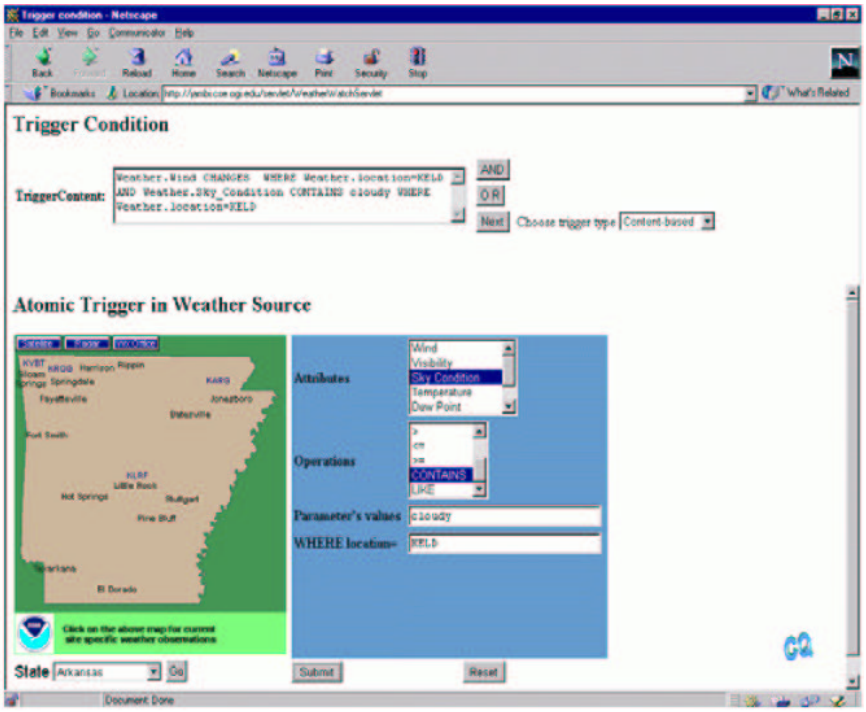

Figure 2: Continual Query Installation Over the Web: An example

Once a continual query is entered in the system, it is classified into a specific index partition identified by a trigger pattern signature (for more details on trigger patterns, see Section 2.4. The trigger evaluation manager will generate a pull query per data source for each trigger pattern (such as a pull query over `Stock.price@stockmaster.com` WHERE

symbol = 'MSFT' for the trigger pattern derived from Example 1). Once the change event observer receives a pull query, it first calls the wrapper of the corresponding source to fetch the data objects of interest, and then compute the recent changes as well as the types of the changes before sending its response to the trigger evaluator. When a trigger evaluator receives a change report from the change event observer, it will locate and test the rest of the trigger condition for each CQ contained in the partition. A change notification is sent out to the CQ users only if the data objects discovered by the change event observer satisfy all the rest of trigger condition of a CQ.

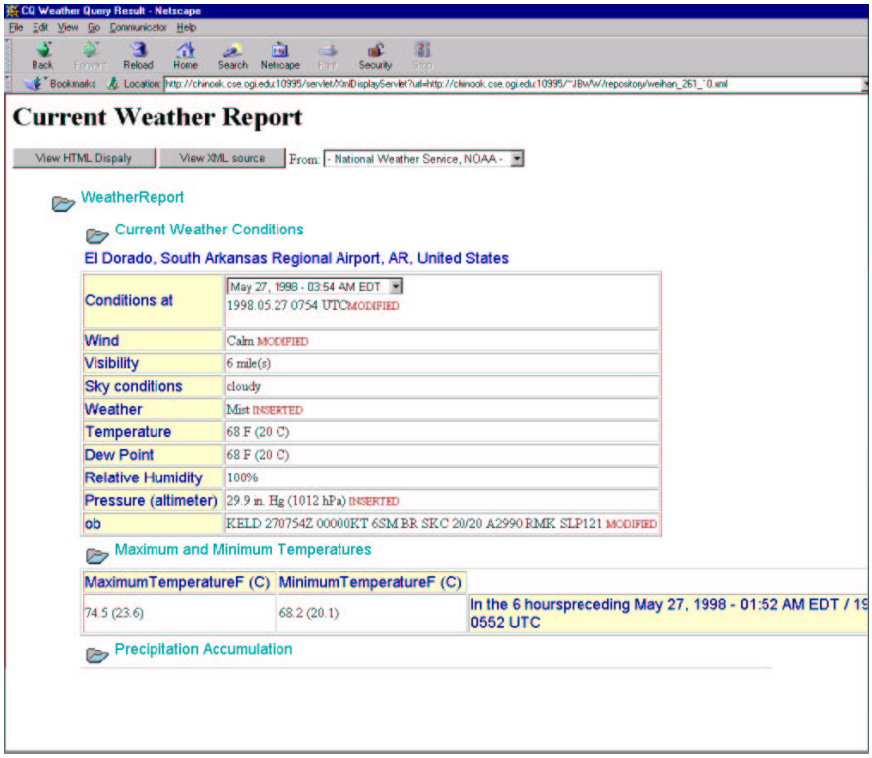

Figure 3: The change report by a change event observer at http://nws.noaa.org

Figure 2 shows an example of a continual query for monitoring the national weather report site (nws.noaa.org) and detecting and notifying content-sensitive changes to the prospective users. Figure 3 shows a differential query result collected in a notification to the specific user. In this case the user is the owner (i.e., the person or application program who installed this continual query). However, OpenCQ has the flexibility to specify notification recipients different from the owner (involving security complications).

2.4 Continual Query Optimization: Grouping by Trigger Patterns

A *Continual Query* has three main components (query, trigger, and notification) for its execution. A CQ trigger is an active component responsible for identifying interesting event happenings at the data sources. The constant fetching of remote information and testing of trigger conditions are the major cost factors in CQ processing.

With thousands of people installing thousands or even millions of CQs in a large-scale information monitoring system, chances are that a lot of monitoring requests may share common components. For example, they could be monitoring the same information (*same trigger, same query*); They could watch the same information source, but have different reactions to the changes (*same trigger, different queries*); They could also have overlapping interests (*partial match for trigger expression*), e.g., one user is interested in IBM and Microsoft's stock price changes while another is monitoring both IBM and Nokia's stock prices.

The main idea behind continual query grouping based on trigger patterns is that many of the trigger expressions share the same or similar structures. Therefore, we adopt an idea similar to inverted files and poll information sources for new information only once for all the CQs that monitor that source. This way, the monitoring cost for the CQ server and data sources remain constant, no matter how many users want to monitor the new information from that source.

The trigger grouping mechanism consists of three main steps. First, all CQs upon their entry to the CQ system will be transformed into a canonical format. Such canonical transformation will be fed into the trigger pattern discovery step. In the second step, one of three alternative algorithms of varying granularity is used. Each CQ trigger is classified into a particular group according to its extracted trigger pattern. In the third step, the CQ runtime engine will make use of the group information for trigger condition testing and evaluation. In the rest of this section we discuss these three steps in detail.

2.4.1 Canonical Transformation

Canonical transformation of trigger expressions is to identify rules to map triggers to trigger patterns. This step is necessary for the grouping based on trigger patterns in a later stage of CQ processing.

As shown in Section 2.1, trigger conditions of continual queries have a common structure consisting of three clauses:

- **AT SOURCE:** refers to one or more source object classes, and some may be suffixed with their data source URLs.
- **WHEN:** consists of a Boolean-valued expression. For each combination of one or more objects referred in the data sources of the *AT SOURCE* list, the *WHEN* clause evaluates to true or false. Each subexpression contained in *WHEN* clause takes the form of *EventObject Operator Value*, where *EventObject* refers to the object being monitored.
- **WHERE:** consists of a logical expression, specifying the context where the *WHEN* condition should be met.

While the *AT SOURCE* clause is simple and intuitive, we transform the *WHEN* clause and the *WHERE* clause into a canonical representation for later grouping. The representation consists of two parts: *Object Event* (WHEN clause) and *Event Context* (WHERE clause). The process is described as follows:

1. Translate the `WHEN` clause to a conjunctive normal form (CNF), i.e., the and-of-ors notation. Each conjunct refers to one or more object classes. We denote each CNF by

$$(C_{11} \vee ... \vee C_{1n_1}) \wedge (C_{21} \vee ... \vee C_{2n_2}) \wedge ... \wedge (C_{q1} \vee ... \vee C_{qn_q}),$$

   where each C_{ij} $(1 \leq i \leq q \; and \; 1 \leq j \leq n_q)$ denotes an event of the canonical form "A_{ij} `op` v_{ij}" or "A_{ij} `op` B_{ij}", v_{ij} is a constant value in the domain of A_{ij} (domain of the event object attribute), and `op` is a string or algorithmic comparison operator, depending on the type of A_{ij}. Domain of B_{ij} is compatible with the domain of A_{ij}.

2. Translate the `WHERE` clause to a conjunctive normal form, similarly to the previous step. Move all conjuncts containing non-equality predicates to the *WHEN clause* by connecting the two parts with "$\wedge$" (logical AND).

3. Group the conjunctive terms by the set of object classes and data sources they refer to in WHEN clause. Let CNF_i denote $C_{i1} \vee C_{i2} \vee ... \vee C_{in_i}$ $(1 \leq i \leq q)$. The result of grouping CNF_i can be represented by

$$(CNF_{11} \wedge ... \wedge CNF_{1h_1}) \wedge ... \wedge (CNF_{m1} \wedge ... \wedge CNF_{mh_m}),$$

where $\sum_{i=1}^{m} h_i = q$.

If a group of conjunctive terms refers to one object class, then we consider the logical AND of these terms to form a *selection predicate group*. If such a group refers to two or more object classes, then we consider the logical AND of its terms to be a join or an n-way-join predicate. These join predicates may or may not contain constants.

4. If the entire `WHEN` expression has p constants, they are numbered 1 to p from left to right. Consider the constant number $k, (1 \leq k \leq p)$. If it appears in C_{ij} in the original expression and C_{ij} is of the form "A_{ij} `op` v_{ij}", then the number k constant v_{ij} in C_{ij} is substituted with the generic term $\texttt{CONSTANT}_k$. If a global list of constant values is maintained in the system, $\texttt{CONSTANT}_k$ can be further replaced with the identifier in the global constant list.

5. If the entire expression is a selection predicate group and it has q operators, they are numbered 1 to q from left to right. For the operator number h $(1 \leq h \leq q)$ if it is the substitute of v_{ij} in original expression C_{ij} of the form "A_{ij} `op` v_{ij}", the operator `op` is substituted with the generic term $\texttt{OPERATOR}_h$.

The output of canonical transformation is a list of rewritten CQs. The system maintains several hash tables in memory. They are for sources, event objects, operators, constants, and WHERE predicates. This is necessary for fast identification of trigger groups in the grouping step. An example output for canonical transformation of 3 CQs is shown in Figure 4.

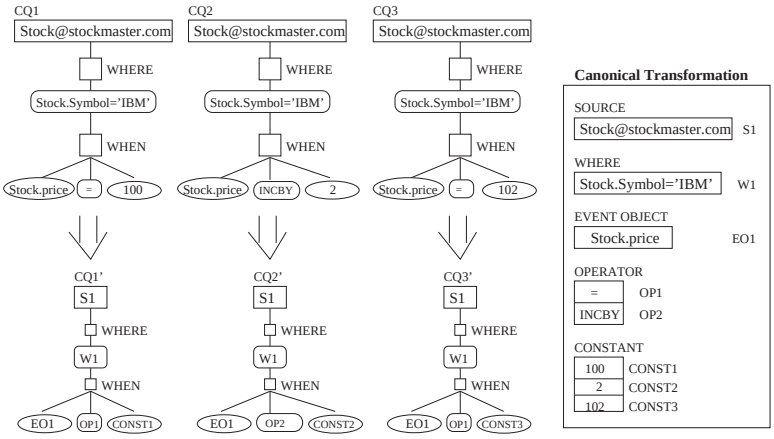

Figure 4: Canonical transformation of continual query

As we can see, the canonical transformation produces compact forms of continual queries for faster matching. Common structures are easy to be identified and grouped together at later stages. For example, CQ1' and CQ3' share most parts of the canonical forms except for the constant part.

Once the canonical form of the original trigger expression is constructed, it is possible for the CQ system to apply one of the trigger grouping algorithms described in the following section.

2.4.2 Alternative Grouping Algorithms

2.4.2.1 Types of Subscription Groups

The main idea behind CQ trigger grouping according to trigger patterns is based on the premise that a large number of triggers often share some of the predicate variables but may take different constant values or different operations in their trigger condition testing. When many triggers are on the same data source, the cost of fetching remote data can be reduced if we group the accesses to common objects together. Currently, we only consider grouping trigger expressions with the same event context[5].

According to the granularity of trigger pattern grouping, we develop the grouping strategies based on 3 types of trigger patterns:

[5]Most Web sites organize data in a page-oriented manner according to the *event context*. For example, you may not be able to ask the stock prices of IBM and Intel at the same time.

- S-type: short for *Simple-type*.
- C-type: short for *Constant-type*.
- OC-type: short for *OperatorConstant-type*.

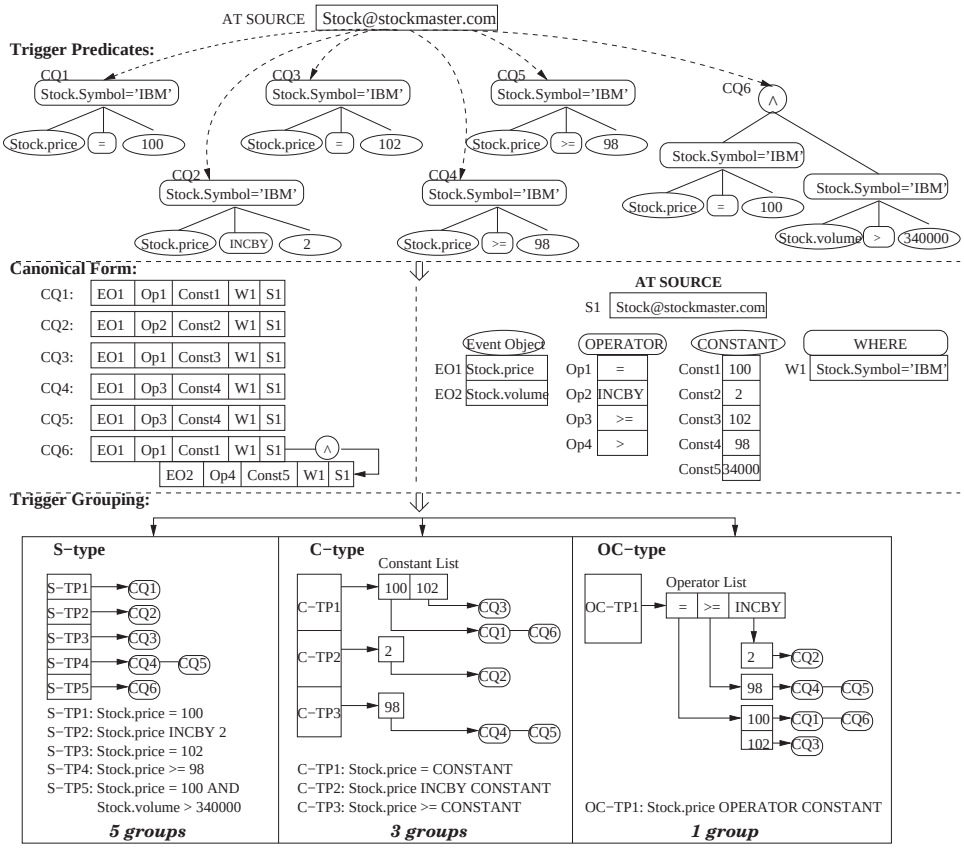

Figure 5: Examples for 3 types of trigger grouping

Figure 5 gives an example of how triggers of different continual queries are grouped according to the three types of grouping algorithms. The triggers of CQ_1-CQ_6 are on the same source (stock@stockmaster.com). After the canonical transformation, a trigger pattern is identified as "Stock.price OPERATOR CONSTANT", with the operators and constants numbered accordingly.

S-type trigger grouping is the simplest trigger grouping strategy, that is, we group continual queries having the exact trigger pattern (same data source, trigger object, trigger context, and operator). This happens when users have the same monitoring specification

over the data objects. This type of grouping has the finest granularity among all types of grouping algorithms since it requires exact match of trigger expressions for CQs in the same group. Therefore, it results in smallest group sizes. An example of an S-type pattern is *"Stock.price = 100"*, which is a simple predicate on the stock source.

C-type trigger grouping relaxes the grouping algorithm on the constant matching in the trigger pattern. For example, when two uses are monitoring IBM stock information, with one set the trigger on "Stock.price = 100" and the other on "Stock.price = 102", they are grouped together since they can share the same change detection query (on IBM's price). The trigger pattern instance is *"Stock.price = $CONSTANT"*. Trigger expression testing will resolve the variable binding for $CONSTANT and evaluate the trigger expression.

OC-type trigger grouping relaxes on both the operator part and constant part in a trigger pattern when considering grouping. That is, triggers having the same change detection query but different operators and values are grouped together. In this case, all 6 CQs are grouped together in one group[6]. The trigger pattern instance is *"Stock.price $OPERATOR $CONSTANT"*.

2.4.2.2 Algorithm Overview

The life cycle of a continual query involves several steps: subscription installation, event detection, trigger condition testing, query, and notification, where query and notification can be optional. CQ subscription installation includes making the continual query a persistent object, canonically transform it, identify the trigger patterns, and classify the CQ into a particular group. Event detection is a ever-running process, which is to detect changes happening at the data sources. Trigger condition testing can also be called trigger expression evaluation. Query gets executed once the trigger condition testing returns true. Finally, notification is activated when query is finished (for *punctual queries*) or when query result is different from the last snapshot (for differential queries).

The CQ subscription clustering (a.k.a CQ grouping) algorithm is described in Table 1.

[6]Note, for CQ4, the trigger has two predicates, one is "Stock.price = 100", the other is "Stock.volume > 34000". Based on selectivity, we choose the first predicate for grouping so that the second predicate is the non-indexable part.

Table 1: Algorithm of Continual Query Grouping by Trigger Patterns

```
INPUT:
        s = [Q, T, N, S_tart, S_top]
        C = {existing subscription clusters}
        L_c = {global constant list}
        L_op = {global operator list}
BEGIN:
        t = canonical_transform(T)
        G = get_cluster_type()
        // Simple (S), Constant (C), or OperatorConstant (OC) type
        case G of
                'S':
                        // nothing
                'C':
                        update_list(L_c, get_constant(t))
                'OC':
                        update_list(L_c, get_constant(t))
                        update_list(L_op, get_operator(t))
        // get the trigger pattern under current clustering algorithm
        p = extract_pattern(t, G)
        if ((c_k = match_cluster(p, get_pattern(C))) == FALSE)
                then {
                        // find new trigger pattern
                        c' = make_cluster()
                        insert(c', s)
                        C = C ∪ c'
                } else {
                        insert(c_k, s)
                }
END.
```

The trigger evaluation of a continual query is divided into the following steps:

1. Selecting grouping type

   Based on the system parameter, we could choose among S-type, C-type, and OC-type of trigger pattern grouping. Currently, only one trigger grouping strategy can be applied at one time. Based on system load and deployment considerations, we can switch between different grouping strategies.

2. Canonical transformation of CQ subscriptions

   This is discussed in the previous section.

3. Trigger pattern recognition

   In this step, all trigger patterns are recognized, with unique pattern ID (signature) assigned to each pattern. CQs having triggers of same pattern will be put into the same group. The system maintains a trigger pattern list. If a new pattern is identified,

a new entry is inserted into the list, with a pointer to the new subscription. Trigger patterns can be classified into S-pattern, C-pattern, and OC-pattern according to the grouping strategy we choose. The global data structures for the grouping algorithm is shown in Figure 5.

4. Event detection query generation

   Each trigger in CQ subscription is decomposed into two parts: *event detection* and *event testing*. For example, if a trigger reads "AT SOURCE Stock@stockmaster.com WHEN Stock.price DECBYP 5 WHERE stock.symbol='MSFT"', the event detection query will be constructed to fetch the remote page on stockmaster.com for the value of Microsoft's stock price. The testing will be simply comparing this current value against '5' with operator *DECBYP*. Detection queries accessing the same page will be grouped together in this step for reduced network delay.

5. Trigger expression testing

   Once the individual event testing is done, the CQ trigger expression consisting composite events is tested and the result is either *true* or *false*. Query component is only activated when the trigger expression is evaluated to be *true*.

2.4.2.3 Implementation Consideration

The CQ trigger grouping algorithms use a set of indices: data source hash table, trigger predicate index for each source, trigger pattern lists, constant list, and operator list. The data source hash is for fast matching between source names and sources IDs. For each source, a predicate index is maintained, indicating predicates on the particular data source. Every predicate in turn will be mapped to specific *trigger patterns* (TP). Based on the grouping strategy chosen, for each trigger pattern containing one or more generic constant terms may need a separate *Constant List* created that links to entries in the *trigger expression* (TE) list (C-type trigger grouping). For OC-type trigger grouping, an additional *Operator List* will be created for every entry in the trigger expression list, with each entry in the operator list pointing to a separate constant list. Figure 6 illustrates the related data structures.

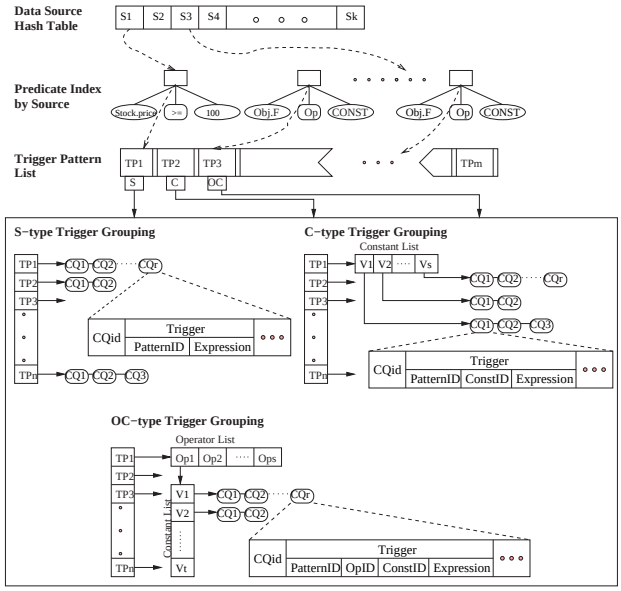

Figure 6: Illustration of CQ trigger grouping algorithms

When a continual query installation request is processed, a number of steps must be performed to update the system catalogs and CQ index structures, and prepare the trigger of the CQ to be ready to run. The primary table in the catalog is the `CQ_Info` table:

```
CQ_Info(cqid,cqName,Query_text,Trigger_text,Start_cond,Stop_cond,creationDate,status, ...)
```

Most of the fields are self-explanatory. The field `status` is used to indicate whether a continual query is currently active. The installation of a continual query is completed after it is successfully recorded in the `CQ_Info` table.

When a continual query is successfully installed and the trigger pattern is extracted, new trigger pattern signatures detected are added to the following table in the CQ system catalog:

```
TrigPatternSig(sigID, objClassID, pattern_desc, constOpTableName, constOpTableSize, ...)
```

The `sigID` field is a unique ID for a trigger pattern signature. The `objClass` field indicates the source object classes on which the trigger pattern is defined. The `pattern_desc` field is a text field with a description of the trigger pattern expression. The constOpTableName field gives the name of the constant-operator table for a trigger pattern signature.

The `constOpTableSize` field gives the number of constant-operator pairs appearing in the trigger pattern expression for the given signature.

When a trigger pattern signature G is derived from the trigger expression E at the continual query installation time, the trigger expression E is broken into two parts: the indexable part and the non-indexable part. The non-indexable portion is NULL when the entire trigger condition is indexable. The format of the constant table for the trigger pattern signature containing N distinct constants is described as follows:

```
ConstOpTable_Num(indexable_exp_id, cqid, indexable_exp_desc, constOpHolder1,...,
                 constOpHolderN, nonIndexablePart_desc)
```

For each trigger pattern signature G, N is determined by the value of `constOpTableSize` in the `TrigPatternSig` record identified by G. The symbol `Num` attached to the name of a constant table is the ID number of the trigger pattern signature. There is a row of the table `constOpTable_Num` for each continual query identified by `cqid`. The key components of each row includes the expression ID of the indexable part of the trigger expression, the unique ID of the CQ containing the indexable expression, the indexable expression description in text, the operator-constant pairs found in the indexable part of the trigger expression of this CQ, and the remaining portion of the trigger condition that is non-indexable.

In short, whenever a trigger pattern is derived from a newly installed CQ, the system will check to see if its signature can be found in table `TriggerPatternSig`. If the trigger pattern is new, it will be added into the table. If this signature has at least one generic constant term, a constant table is created for this trigger pattern signature with a entry added to record the indexable part, the list of constant-operator pairs, and the non-indexable part of this newly installed CQ.

2.4.3 Index on Most Selective Atomic Trigger Patterns

In Section 2.4.1, every trigger condition pattern is a CNF. The simplest trigger pattern is a CNF with a single conjunct containing no OR operators and has the form "`[<objClassName>.]<fieldName> <comparisonOp> <CONSTANT>`". We call such a trigger pattern an *atomic trigger pattern*.

For convenience of discussion, in this section we consider each trigger pattern `G`, produced from the canonical transformation of the original trigger expression, to be a selection predicate group, consisting of a set of atomic trigger patterns. We delay the treatment of OR predicates and join predicates to later sections.

Let a trigger pattern be denoted by $AC_1 \wedge AC_2 \wedge ... \wedge AC_m$, where each AC_i $(1 \leq i \leq m)$ is of the form "F_i op_i `CONSTANT`$_i$" and F_i denotes a filed name possibly with a class name referenced in the `FROM` clause as the prefix (e.g., `Stock.price` in Example 1).

When a continual query is routed to a single source, this selection predicate group can be processed together. When the continual query is routed to a set of n data sources, then the CQ trigger manager will spawn n threads, where each thread processes the selection predicate group at a single source.

For each given trigger pattern `G`, the next decision is to select the atomic trigger pattern that is most selective to index. More concretely,

- If $n = 1$, then AC_1 is the only conjunct and it is chosen as the indexable predicate.
- If $n > 1$, then a single conjunct AC_k is identified as the most selective one and thus the indexable part of `G`, if and only if $\forall AC_i (1 \leq i \leq m), SF(AC_i) \geq SF(AC_k)$.

All trigger expressions clustered to the group G are indexed by AC_k directly. $SF(P)$ is a selectivity factor of predicate P. In general, a predicate of the form `attribute_name = constant` has the lowest selectivity factor. The predicate with one of the range comparison operators $(\leq, <, \geq, >)$ is higher and the predicate of the form `attribute_name <> constant` has the highest SF.

The rest of conjunctive terms are located and tested only if a data object fetched from the remote data source(s) matches the most selective term. When the remaining terms in the trigger condition also match, we sat that the data object is completely matched the trigger condition.

Using the alternative indexing approach based on the most selective atomic trigger patterns of each CQ, the five continual queries will all be eligible to be grouped by indexing on the atomic trigger pattern "`Stock.price OPERATOR_1 CONSTANT_1 WHERE symbol`

= 'IBM'". Thus, cq1, cq2, cq3, and cq5 are clustered into one group and are processed through a single remote access to the relevant data sources, instead of one remote access for each CQ.

2.4.4 Optimization of Other Complex Trigger Expressions

A potential topic for future work is to optimize the processing of selection predicates containing OR's, joins, or very expensive functions [45]. The main idea is to explore optimization opportunities inherent in the particular properties of each type of operators. Consider the trigger conditions that contain OR's. If a single one of the OR'ed clauses is true, then the entire predicate is true. By properly ordering the processing of OR'ed clauses, this observation will help in speeding up the evaluation of the predicate with ORs. However, this optimization helps little when all OR'ed clauses are not true. Our initial proposal for handling OR predicates is to index each atomic OR'ed clause whenever the number of CQs sharing the same atomic trigger pattern reaches certain system-defined threshold.

We are also working on the strategies for offering both memory-based and disk-based data structure as alternatives for organization and utilization of CQ indexes. The main idea along this line of work is to build on Rete [34] and TREAT [74] algorithms for efficient implementation of AI production systems. It is known that Rete and TREAT algorithms make the implicit assumption that the number of rules in AI rule systems is small enough to fit in main memory. A main challenge is to develop strategies and mechanisms that can automatically manage the utilization of memory-based or disk-based CQ indexes at runtime.

2.5 Experiments and Results

2.5.1 Experiment Setup

The experiments were done on a combination of the prototype OpenCQ engine with a simulated CQ system for scalability experiments. The simulator uses the same code from the prototype OpenCQ implementation, but it is able to "run" a much higher number of CQs than native OpenCQ. The result is that we prepared a synthetic workload for most of experiments. The time for installing CQs and loading them into memory is not counted

in the total running time. Every experiment was run 10 times and the average run-time result was computed to mitigate the effect of initialization cost (e.g., system cache fill-up and Just-In-Time compilation of Java methods) and Java garbage collection.

The system hardware consists of a Sun Enterprise 450 server with 4 400-MHz Ultra SPARC-II processors, 1GB of memory, enough disk space, and a 100Mbps Ethernet local area network. The system run-time software consists of Oracle 9i Enterprise DBMS (Release 9.0.1) running on Solaris 7. The host language consists of Java-2 run-time environment (JDK1.3) with Java Hotspot client VM.

2.5.2 Workload

The test data were collected from <Yahoo!> and <http://www.dbc.com>[7] at the beginning of February 2000. Stock symbols were extracted from Yahoo! Web pages and quotes were extracted from DBC.com using XWRAP Elite wrapper tools [118]. Two simulated data sources were set up (Stock, quotes over 8337 symbols, and StockNews, with news information about companies represented in the set of stock symbols) to satisfy the requirements of scalability experiments in the simulator. The simulated data sources reside in local area networks to avoid the fluctuation in access time when fetching remote data. For simplification of the evaluation, instead of sending out emails to users, the notification components are simply writing the notification results in a log file at the server.

We studied four trigger types in our experiments. Type 1 trigger has an arithmetic comparison operator, including $<$, $\leq$, $\geq$, $=$, and $\neq$. Type 2 trigger has a CQ system built-in cache operator, including INCBYP (*increased by percentage*, which compares the previous query result and the current database state) and other similar operators: DECBYP, INCBY, and DECBY. Type 3 trigger has multiple detection conditions, connected with relational operators "AND" or "OR". Type 4 trigger is essentially a trigger for detecting changes on multiple data sources (like "join" operator for database tables).

- Type 1 trigger (arithmetic comparison operator) $\rightarrow$

  Example: tell me IBM's stock quote whenever its last trading price is greater than $110.

[7] DBC.com has moved to eSIGNAL.com at the time of writing this paper.

Trigger: AT SOURCE stock@dbc.com WHEN stock.last > 110 WHERE stock.symbol = 'IBM'

Query: EXTRACT ALL FROM stock@dbc.com CONDITION symbol='IBM'

- Type 2 trigger (cache comparison operator) →

  Example: tell me IBM's stock quote whenever its last trading price is changed by 5%.

  Trigger: AT SOURCE stock@dbc.com WHEN stock.last INCBYP 5 WHERE stock.symbol = 'IBM'

  Query: EXTRACT ALL FROM stock@dbc.com CONDITION symbol='IBM'

- Type 3 trigger (multiple conditions) →

  Example: tell me IBM's stock quote whenever its last trading price is over $108 and the volume is greater than 200000.

  Trigger: AT SOURCE stock@dbc.com WHEN stock.last > 108 WHERE stock.symbol = 'IBM' AND stock.volume > 200000 WHERE stock.symbol = 'IBM'

  Query: EXTRACT ALL FROM stock@dbc.com CONDITION symbol='IBM'

- Type 4 trigger (join operator) →

  Example: tell me Nokia's stock quote whenever some telecommunication companies appear in the news headlines and whose last trading prices are over $45.

  Trigger: AT SOURCE stock@dbc.com WHEN stock.last > 45 WHERE stockNews.title LIKE '%Telecommunication%' ∧ stockNews.stocksymbol = stock.symbol

  Query: EXTRACT ALL FROM stock@dbc.com CONDITION symbol='NOK'

Although the four types of triggers differ in terms of evaluation time, they have similar running characteristics. Therefore, when presenting the experimental results, if not stated, the analysis applies to all types of triggers. The simulation workload consists of triggers with their types evenly distributed. For more motivational examples for trigger grouping, please refer to [62, 68, 69] for more details.

For CQ grouping, we study two types of workloads in terms of group sizes: *fixed* and *skewed*. For fixed workload, incoming CQs are uniformly distributed among groups of equal size after applying the CQ grouping algorithm. For skewed workload, the group sizes are distributed according to Zipf-like distributions after the classification algorithm. Zipf's

law [125] states that the frequency of some event P, as a function of a rank i, is a power-law function with P_i proportional to $1/i^{\alpha}$ (α=1 in strict Zipf's distribution [17]). In our experiments, the probability density function used for general Zipf-like distribution is shown in Equation 1 below:

$$P(i) = C/i^{\alpha} \tag{1}$$

where C is a normalizing coefficient.

The intuition behind Zipf-like distribution is that a large number of users have common interests (sharing a small number of CQs on the same subjects), a large number of CQs with a small number of users interested in each, and a moderate number (less than linear) of users for CQs in-between. Previous research has shown that the distribution of Web page requests follows Zipf-like distribution [17, 122], with α varying from 0.5 to 1.24. For generality, we choose the range of α values of the Zipf-like distribution to be [0.4, 1.6] in our experiments.

2.5.3 Performance Evaluation Model

The simulator used the parameters in Table 2 for scalability experiments:

Table 2: OpenCQ Performance Evaluation Parameters

Parameter	Description	Value
SZ_{Stock}	Stock data size	8337 records,834KB
$SZ_{Stocknews}$	Stocknews data size	45772 records,9393KB
BD	Network bandwidth	100Mbps
N_{cq}	total number of active CQs	200-60000
TF	percentage of triggers fired	100%
QF	percentage of queries fired	0-100%
NF	percentage of notif. fired	0-100%
N_G	number of groups	1-N_{cq}
TC_g	grouping cost	50-200ms/group
H_Q	query cache hit ratio	20%-40%
α	Zipf popularity parameter	0.4-1.6

In the performance evaluation, we measured the system performance in terms of elapsed time and system throughput. CQ elapsed time is measured as the time difference between starting and ending of the evaluation of a batch of continual queries in the system. System throughput is presented in number of CQs evaluated per second, which is calculated by dividing the total elapsed time of evaluating a batch of CQs by the number of CQs in the

batch.

For a batch evaluation of a set of continual queries, the simulation models of elapse time for both non-optimized and optimized cases[8] are shown in equations 2 and 3.

- Non-optimized case:

$$T^{N_{cq}} = \sum_{i=1}^{N_{cq}} (T_{i_{trigger}} + T_{i_{query}} + T_{i_{notification}}) \tag{2}$$

  where the total elapsed time is the summation of sequential execution of each CQ component.

- Optimized case:

  The total elapsed time $T^{N_{cq}}$ for evaluation a batch of continual queries is the sum of evaluation times for triggers, queries, and notifications as shown in equation 3. We assume that every continual query belongs to a group.

$$\begin{aligned} T^{N_{cq}} &= T_{trigger} + T_{query} + T_{notification} \\ &= \sum_{i=1}^{N_{G_{trigger}}} (T_{i_{trigger}} + TC_{trigger}) + \\ &\quad \overline{\tau_{query}} \times (N_{cq} \times QF) + \\ &\quad \sum_{j=1}^{N_{G_{notif}}} (T_{j_{notif}} + TC_{notif}) \end{aligned} \tag{3}$$

  where

  - $N_{G_{trigger}}$ and $N_{G_{notif}}$ are the numbers of groups for trigger and notification, respectively.
  - $T_{i_{trigger}}$ and $T_{j_{notif}}$ are the respective elapsed times for evaluating the trigger and notification components.
  - $TC_{trigger}$ and TC_{notif} are the cost related to accessing group membership information for trigger and notification respectively.

[8] We also refer to "non-optimized" as "non-grouping" and "optimized" as "grouping".

– $\overline{\tau_{query}} = \tau_{query_{hit}} + (1 - H_Q) \times \tau_{query_{miss}}$, where $\tau_{query_{hit}}$ is the evaluation time when a query hits memory cache, which is close to zero because we assume cost of memory access is trivial compared to remote query cost. H_Q is the query cache hit ratio. $\tau_{query_{miss}}$ is the cost for remote query result fetching.

– QF is the query-firing ratio.

The ultimate goal is to increase the overall system throughput for evaluating all installed continual queries without causing too much delay in response time for individual CQs. We also show the benefit of grouping in terms of the speedup ratio to the baseline non-grouping CQ evaluation. The result is shown in the next section.

2.5.4 Performance Results

To get a "typical" CQ execution time, we show the performance breakdown for the evaluation of a typical CQ (although the evaluation times differ according to different types of triggers, queries, and notifications) in the following table (measured for Type 2 trigger on average of 14000 data points):

	Trigger	**Query**	**Notif.**	**Bookkeep**
Mean Elapsed time	42.8ms	61.1ms	7.3ms	11.4ms
Standard Dev.	18.8	17.4	1.8	10.3

We organize the rest of this section as follows: Section 2.5.4.1 studies the performance characteristics of different types of triggers and benefit of grouping with fixed group size. Section 2.5.4.2 studies the impact of data source size change. Section 2.5.4.3 discusses the effect of fixed group size workload on trigger grouping performance. Section 2.5.4.4 discusses the effect of skewed group size workload on trigger grouping performance. Finally, Section 2.5.4.5 analyzes various sources of grouping costs.

2.5.4.1 Varying Triggers and Fixed Group Size

The benefit of CQ trigger grouping is shown in Figure 7.

In each of the graphs, we demonstrated the benefit of CQ trigger grouping with varying group sizes compared with the non-grouping case. The figure shows clearly that trigger grouping can cut the evaluation time in orders of magnitude according to the group sizes.

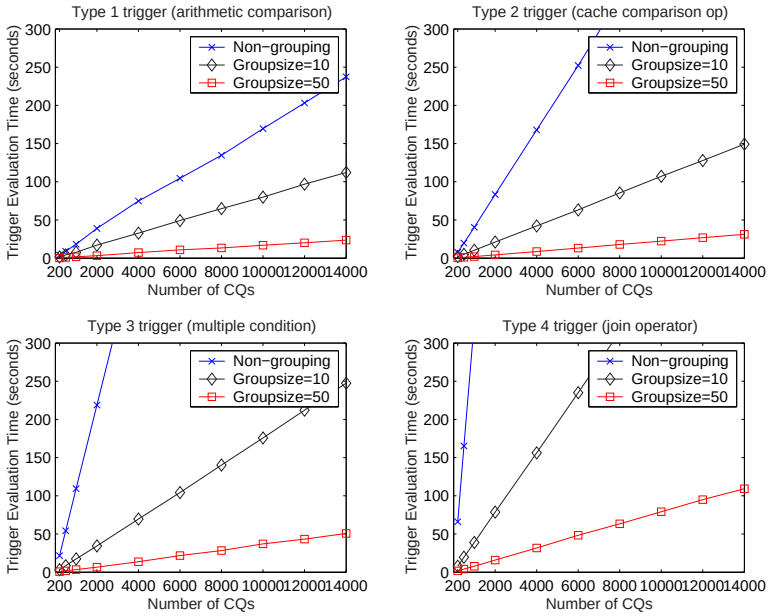

Figure 7: Grouping benefit for different types of triggers

When group size is larger, the trigger evaluation time is shorter. This is because we only need to evaluate the common part of multiple trigger once for a group of CQs. The total evaluation time is linear to the number of CQs in the system. The execution time saved is proportional to the group size increase. For example, for trigger type 1, when group size increases from 10 to 50, the evaluation time dropped from 94.7 seconds to 18.4 seconds. However, group size of 10 does not mean the grouping evaluation can get to one tenth the cost of non-grouping evaluation. This is due to the presence of grouping cost, which is mainly the time needed to access various grouping indices. Grouping cost will become minimal once we preload all the index tables into main memory. In Section 2.5.4.5, we will study grouping cost in details.

Readers familiar with the CQ literature may have noticed that Figure 7 reproduced the main results from Niagara [25]. To the best of our knowledge, Niagara experiments used a single query (which contains only the trigger evaluation) under two conditions (no grouping, and all queries always in one group).

In our case, Figure 7 also demonstrates that different triggers have different evaluation costs (higher cost triggers have steeper performance curves). Triggers with join operator are generally more expensive than other types of triggers. This is because this type of trigger

involves fetching data from multiple data sources, which is generally more expensive than matching predicates in main memory for trigger evaluation.

2.5.4.2 Impact of Data Size

In this experiment, we study how different data sizes affect the evaluation of continual queries. Two types of triggers are studied: type 1 trigger discussed in the previous section, and trigger with an aggregation function (an example is shown below).

Example: tell me IBM's stock quote whenever the average stock trading volume exceeds 340000.

CQ Trigger: AT SOURCE stock@dbc.com WHEN AVG(stock.volume) > 340000

CQ Query: EXTRACT ALL FROM stock@dbc.com CONDITION symbol='IBM'

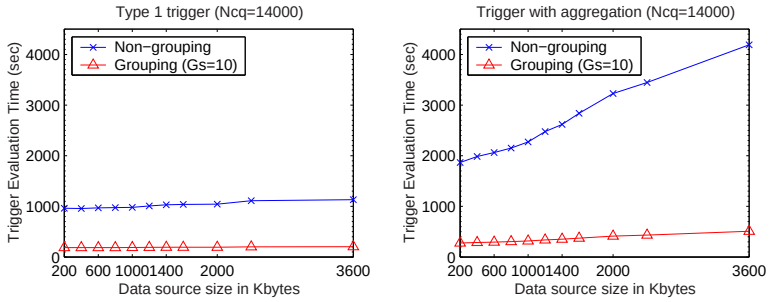

Figure 8: Data Size Impact on Trigger Grouping

Figure 8 compares non-grouping and grouping cases for the two types of triggers when data size changes in terms of elapsed time as well as throughput. Data size increase brings up the evaluation times for both types of triggers. However, the impact on type 1 trigger is far less significant than that on trigger with aggregation. This can be seen from the two relative flat lines in the upper two graphs. For type 1 trigger, since the triggering attribute is indexed, it is not very sensitive to data size change. While for triggers with aggregation, data size increase results in an sharp increase in trigger evaluation time. However, the data size change does not affect the performance under grouping significantly. With group size of 10 CQs/group, the throughput for the grouping engine is roughly 10 times that of the non-grouping engine (when we do not consider grouping cost). The system performance is more stable using trigger grouping on data sources of different sizes.

2.5.4.3 *Impact of Fixed Group Size*

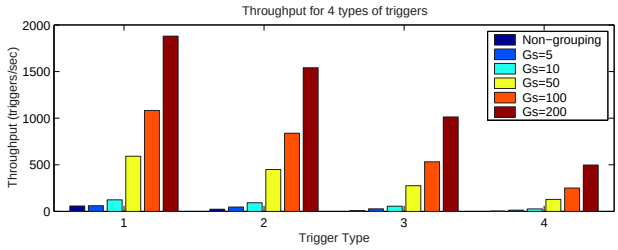

Figure 9: System throughput under fixed group sizes

Figure 9 demonstrates how different settings on the fixed group sizes affect the throughput of CQ trigger processing. As we can see, given a fixed workload on CQ trigger grouping when group size is 200CQs/group, the CQ system can evaluate up to 2812, 2202, 1235, and 541 triggers/second for the four types of triggers. While for fixed group size of 5CQs/group, the numbers are only 74, 53, 30, and 13, respectively. The trigger evaluation time is inversely proportional to the size of the groups.

2.5.4.4 *Impact of Skewed Group Size*

We have presented above the performance characteristics for CQ trigger grouping under the fixed workload. In this section, we study the trigger grouping under the Zipf-like workload.

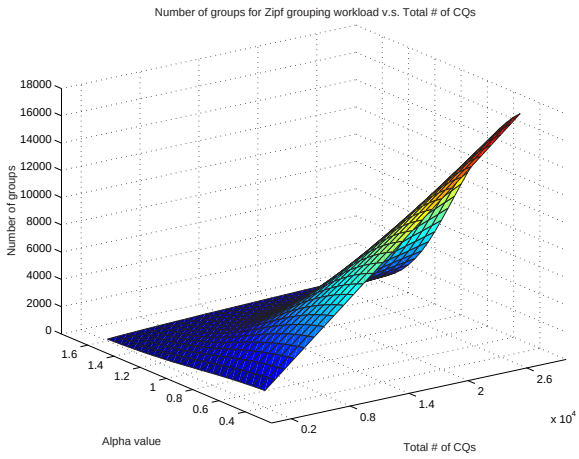

Figure 10: Skewed (Zipf) workload: alpha, number of groups, and total CQs

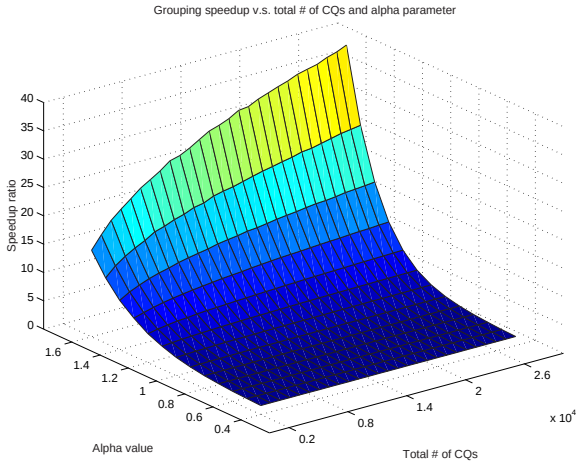

Figure 11: Trigger grouping speedup for skewed (Zipf) workload

For Zipf-like workload, a high α value indicates that popular objects are *very popular*, and unpopular objects are *very unpopular*. It means that the bigger groups are much larger.

Accordingly, a low α value indicates a more homogeneous user subscription stream (user interests are widely spread), with many users interested in many different data objects. The number of groups tends to be higher and group sizes smaller when α value is lower, as described in Figure 10.

Figure 11 shows the speedup for trigger grouping under skewed workload. The speedup is higher when α is higher. This is because a higher α value indicates a more heterogeneous subscription group configuration (larger groups are much larger than smaller ones) Thus there are fewer groups in the system after grouping, which in turn saves more evaluation time.

Figure 12 shows the relative performance of trigger grouping under different configurations of group distributions: fixed and skewed (Zipf). The result indicates that fixed workload, when group size is 2 CQs/group, performs similarly to skewed workload with α parameter of 0.6. And system performance for group size of 5 CQs/group under fixed workload is close to the system performance under skewed workload with $\alpha = 1.1$.

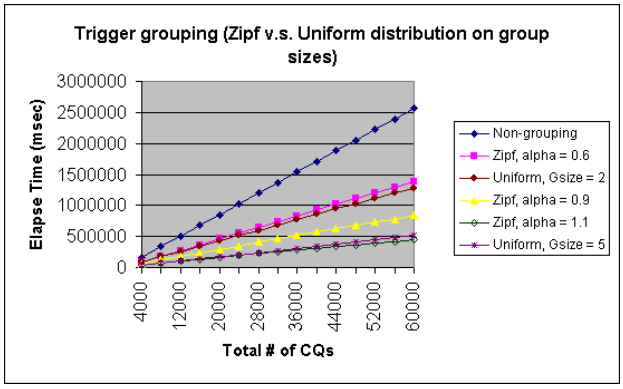

Figure 12: Comparison of trigger grouping under different workloads

2.5.4.5 Costs of grouping

The presence of grouping cost is illustrated in Figure 13 and 14. Figure 13 shows the analytical results for CQ grouping speedup under the uniform workload, while Figure 14 shows the real system measurement. Although analytical study shows the grouping speedup to be equal to the group size, real measurement of the system can only get up to a proportion of the analytical value (e.g., when group size is 180CQs/group, the speedup ratio is only

close to 30). The difference indicates the presence of extra cost using the grouping scheme.

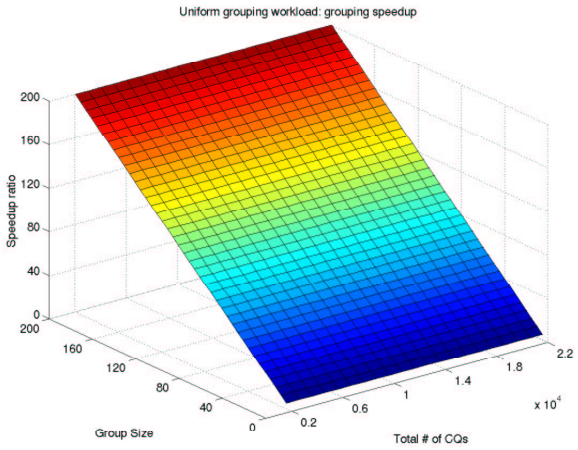

Figure 13: Speedup ratio under uniform workload (analytical)

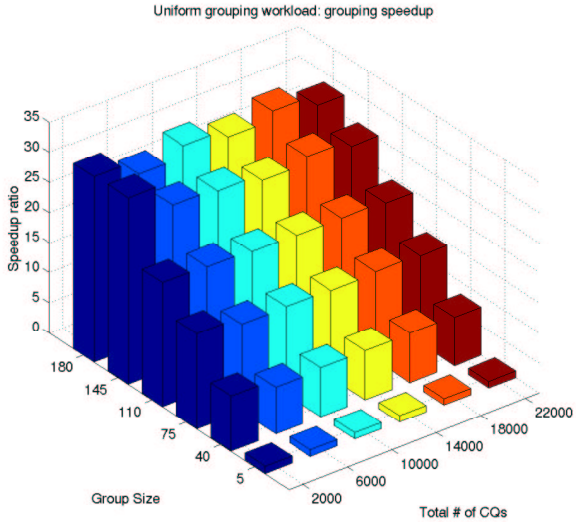

Figure 14: Speedup ratio under uniform workload (measured)

The reason of grouping cost is because extra data structures are needed to store and retrieve the information about CQ group subscriptions. There are three kinds of grouping costs: installation cost, runtime cost, and maintenance cost.

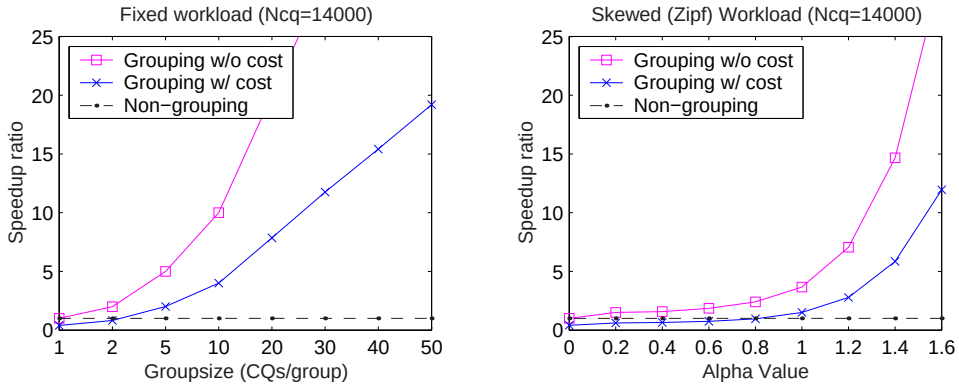

Figure 15: Grouping benefit threshold

- Installation Cost: At installation time, before a new continual query is made persistent, we have to identify the group to which this particular CQ belongs. A new group will be created if necessary. A proper group ID will be assigned to this CQ and associated group data structures will be updated.

- Runtime Cost: At runtime, when CQs are being evaluated, the system has to first

retrieve grouping information from group index tables on the fly. The grouping cost is namely the lookup time for each group index table. In a distributed environment, we cannot assume the information about each group will reside in main memory of one host machine. Extra cost will therefore include networking delays when we retrieve group information from remote hosts. For example, Figure 15 illustrates the trigger grouping speedup ratios in the presence of runtime grouping cost. It can greatly impact the CQ system performance. In the performance study of this paper, the grouping cost refers to this runtime cost.

- Maintenance Cost: We may need to adjust continual query grouping at certain times when system performance degrades. This happens when group sizes change among CQ groups. We observe that when group sizes are too small, CQ grouping does not help the system performance at all. We take trigger grouping for an example. In Figure 15, grouping is not beneficial any more when for situations falling below the dashed baseline. For fixed group size workload, when group size falls below 5 CQs/group, grouping does not benefit at all under grouping cost of 200ms/group. When the group size is smaller than 2 CQs/group, trigger grouping performs worse than the non-grouping case under all three configurations of grouping cost. Similarly, for the skewed workload when group sizes follow Zipf distribution, cost of grouping has to be considered when the grouping performance curves fall below the baseline. We may need to change the current grouping algorithm to a coarser one (e.g., from S-type to OC-type trigger grouping) dynamically.

2.6 Discussions on Multi-level Optimization of CQ

In addition to the CQ grouping technique presented in Section 2.4, we can further apply multi-level optimization techniques in CQ processing at query and notification levels, namely *query result caching* and *notification grouping*.

Query Result Caching:

When multiple queries in CQ subscriptions are on the same data source or remote object, query caching is useful to save the remote fetching time. For example, consider the following

queries for CQ1 and CQ2, in which two users are both interested in the *NASDAQ Composite Index* value:

```
CQ1:
   Query:    EXTRACT daylow, dayhigh
             FROM    stock@stockmaster.com
             CONDITION symbol = 'NASDAQ';
   Trigger:  AT SOURCE stock@stockmaster.com
             WHEN stock.low < 90
             WHERE stock.symbol = 'IBM';

CQ2:
   Query:    EXTRACT daylow, dayhigh
             FROM    stock@stockmaster.com
             CONDITION symbol = 'NASDAQ';
   Trigger:  AT SOURCE stock@stockmaster.com
             WHEN stock.high DECBYP 5
             WHERE stock.symbol = 'MSFT';
```

Although the trigger of the two continual queries are different, the trigger component of the two are the same. We save the query evaluation time by fetching the target page once and make it available in the query cache. Without query result caching, the same page containing the *NASDAQ Composite Index* value has to be fetched twice from the remote Web site.

Currently, query result caching in the continual query system employs an LRU policy for cache replacement. Proxy caching can be employed to reduce the response time for the remote fetching step in query evaluation. While the cache consistency issue has yet to be studied.

Notification Grouping:

In current implementation of OpenCQ system [62, 100], we group notifications according to their destinations (e.g., email addresses). That is, instead of sending to the same user multiple notifications on his/her different CQ subscriptions, the system groups the notification into one summary email and sends to the user once. The advantage is two-fold: 1. individual users do not have to receive multiple emails from the system; 2. the CQ system decreases the number of emails it has to send out thus reduces the server load.

Another scenario for notification grouping is that we can group notifications for users from the same domain by setting up a domain proxy/dispatcher. Notifications for the domain will be delivered by CQ server in batch. Then the domain dispatcher disseminates individual notifications to the end users by utilizing fast local network connections. The proxy/dispatcher is responsible for local resource sharing. In the mean time, other alternatives can be explored including IP-multicast [82].

The multi-level optimization result is shown in Figure 16. With trigger grouping only, system performance is 1.34 times higher than non-grouping case. While by applying all three levels of grouping: trigger grouping, query result caching, and notification grouping, we can achieve up to 2.14 times performance improvement. We choose skewed workload (Zipf $\alpha = 0.9$) for trigger grouping and assume each user has an average of 10 registered CQs in the system when considering notification grouping. This workload is considered to be representative in a continual query system.

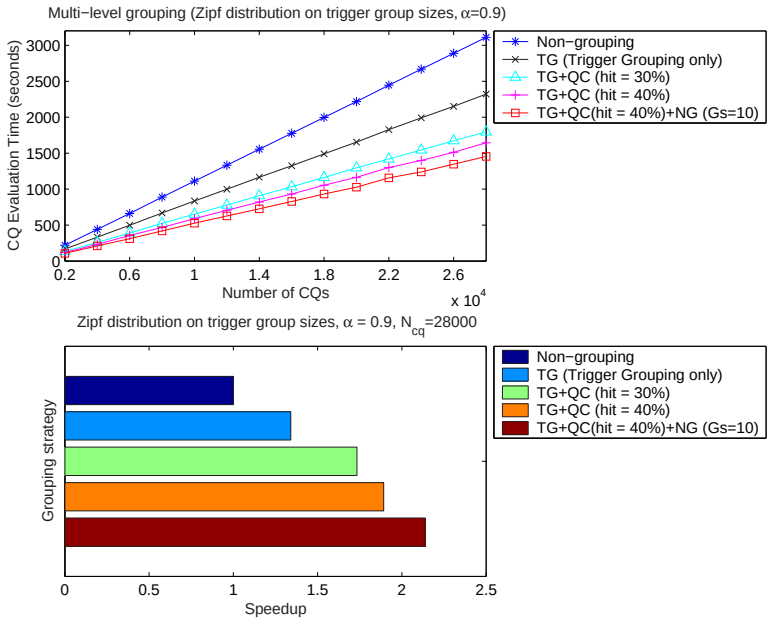

Figure 16: Multi-level Continual Query optimization

The improvement of multi-level optimization on system performance is shown in the following table:

Optimization Scheme	TG	TG+QC(H=30%)	TG+QC(H=40%)	TG+QC+NG
Speedup	1.34	1.57	1.89	2.14

As we can see, multi-level CQ optimization performs better than applying a single grouping algorithm (e.g., trigger grouping alone), In order for a CQ system to achieve the maximum performance gain from grouping, we have to exploit optimization opportunities at trigger, query, and notification levels altogether.

2.7 System Status

The *Continual Query* project was started in 1997 when I was a Master student at the College of Computing in University of Alberta, Canada [99]. The first prototype was implemented in Perl and demonstrated in ACM SIGMOD conference in 1998 [63]. The system was converted to Java after I moved to Oregon Graduate Institute in 1998. The system was made publicly available in summer 1999 and was demonstrated to DARPA funding agencies. The system is now maintained at the College of Computing in Georgia Institute of Technology. You can access the online demo of the OpenCQ system at http://disl.cc.gatech.edu/OpenCQ. The downloadable code can be accessed at
http://disl.cc.gatech.edu/CQ/plugin/download.html.

CHAPTER III

WEB PAGE CHANGE MONITORING

On the World Wide Web, a particular class of semi-structured data is Web pages (primarily in HTML format). In this chapter, we present the specific problems, challenges, and our solutions for HTML Web page monitoring.

3.1 Motivation and Problem Statement

As the Web grows and evolves, we observe some rapid changes in the ways in which fresh information is delivered and disseminated. The mode of data transfer is shifting from a "pull-only" model to a "push-pull" model [2]. Many believe that the "push" style of information delivery and dissemination is, to some extent, a natural solution to the scale of the Internet. In the "push-pull" model, some data is *pushed* to users without an explicit pull request. The "push" style enables services to be served asynchronously as they become available. Instead of having users tracking when to visit web pages of interest and identifying what and how the page of interest has been changed manually, the information change monitoring service is becoming increasingly popular, which enables information to be delivered automatically while it is still fresh. The push service is specially suited for busy individuals and for delivering transient data such as stock quotes, product prices, news headlines, and weather information.

However, designers of large-scale Web-based change monitoring and notification systems face a common problem: HTTP [33] is a pure request-response model and it does not allow servers to asynchronously notify clients of events on the server-side. As a result, search engines to date, although powerful in helping users locating and finding information of interest, do not support tracking changes on behalf of users and cannot deliver timely information to the right users at the right time.

From systems perspective, many distributed systems need the functionality of asynchronous event dissemination. Examples include callbacks in distributed file systems [90] and gossip messages in lazy replication systems [57]. Although several existing systems, such as ICQ [48] and Pointcast [83], support large-scale notification using centralized proxies that relay events from servers to clients, these notification mechanisms are specialized for their applications. Hence, it is desirable to design a general-purpose event dissemination infrastructure on which large-scale change monitoring systems can be implemented.

Now let us look at the problems from users' perspective, namely what is the common behavior of users who wish to monitor changes in web pages. Individuals often use a search engine to find a page of interest, and then bookmark the pages that they wish to re-visit. Upon a revisit, a fresh copy of the page will be obtained, and the user needs to determine if it has changed in an interesting way manually. Obviously the first challenge is the ability to allow users to only revisit a page *when* the page has changed in a way that is interesting. Furthermore, if a user becomes interested in tracking changes over a large number of web pages, he or she may wish to be notified not only when to re-visit the pages of interest but also what the concrete changes are. This is because when the number of pages of interest grows large it will be difficult, if not impossible, for a user to remember the concrete details about every page in which he or she was interested. Therefore, the second important challenge is to identify *what* and *how* the page has changed, including the types of changes and the amount of changes between the fresh copy and the copy last seen.

3.2 The State of Art and Technical Challenges

Several tools are available to assist users in tracking of *when* web pages of interest have changed [30, 15]. Most of these tools are in .com domain and offer tracking service either from a centralized server or a client's machine. Server-based tools track pages that are previously registered or submitted by users and notify of users changes via email or over the Web upon a request. Examples include Netmind [78], TracerLock [104], AIDE [30], and Webclipping.com [108]. Client-based tools run on a user's machine, and track changes either periodically or on demand, such as WebWhacker^{TM}[113]. We can also classify these

tools based on the coverage of the service. Some tools offer tracking service on a specific or a constrained set of URLs instead of on any registered URLs. For example, several client based tools, such as Smart Bookmarks [96] or Bookmark Surfbot [80], use the user's list of bookmarks; other tools either restrict the number of URLs to be monitored for a user, such as WWWFetch [117] or track the new web sites of chosen topics such as WebCatcher [107], or monitor changes of selected topics (e.g., current stock prices, weather forecasts, sports scores, headlines) such as WebSprite [112].

Up till now most tracking tool development has gone on at companies with little exposure of technical details, especially the efficiency, the scalability, and the tracking quality of such systems. Furthermore, from individual users' perspective, we observe three common problems with these tools. First, with the exception of Netmind [78] and AIDE [30], most of the tools only address the problem of when to re-visit a fresh copy of the pages of interest but not the problem of what and how the pages have changed. Second, all these tools handle the *when* problem with a limited set of capabilities. For instance, Netmind can only track changes on a selected text region, a registered link or image, a keyword, or the timestamp of the page. The third problem with all these tools is the scalability of their notification service with respect to individual users. Typically, these tools treat each Web page tracking request as a unit of notification. Users who register a large number of pages with the tracking service are easily overwhelmed with the large number of frequent email notification messages.

3.3 WebCQ System Architecture

WebCQ is a server-based change detection and notification system for monitoring changes in arbitrary web pages. The system consists of five main components as follows (shown in Figure 17):

- a change detection robot that discovers and detects changes to arbitrary Web pages
- a proxy cache service that reduces the communication traffics to the original information provider on the remote server

- a trigger evaluation tool that filters only the changes that match certain thresholds
- a personalized change presentation tool that highlights changes between the web page last seen and the new version of the page
- a centralized change notification service that not only notifies users of changes to the Web pages of interest but also provide a personalized view of how Web pages have changed and what fresh information should be delivered.

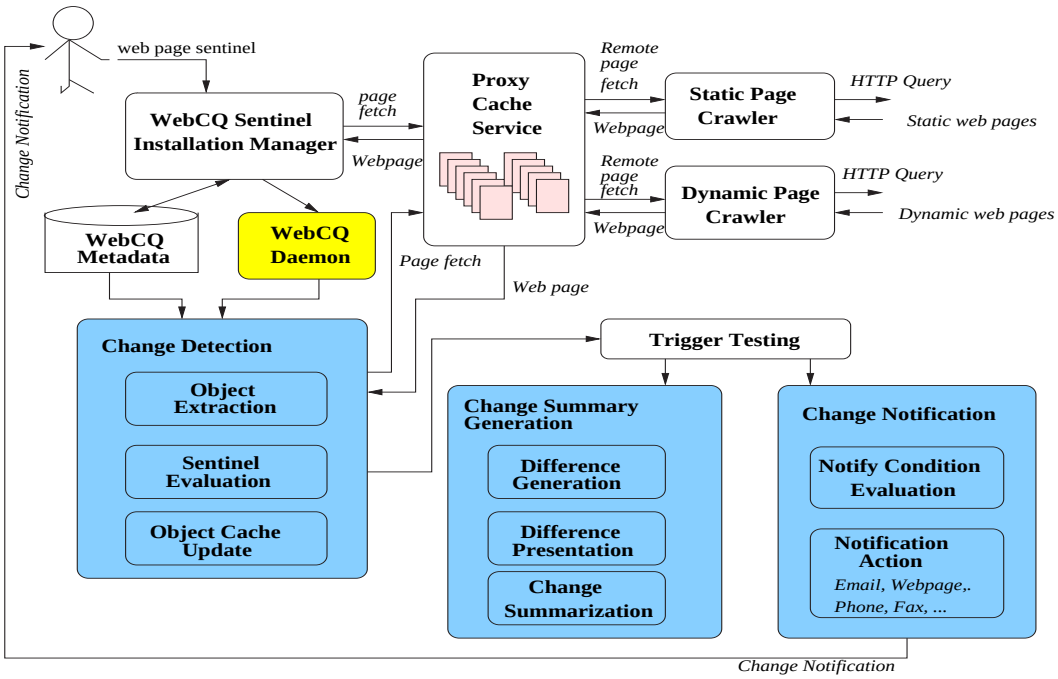

Figure 17: WebCQ System Architecture

In the first release of WebCQ, users register their web page monitoring requests with WebCQ using HTML forms. Typically a user enters a URL of interest (say the Internet movie database). The WebCQ installation manager will pass this URL to the proxy server. The proxy server will display the page in the lower frame of the WebCQ window as shown in Figure 18.

The user can click on any link of interest on the page until she reaches the exact page that she wishes to monitor (see Figure 18). Now the user may select the type of information content she wants WebCQ to monitor for her from the pull-down menu shown in

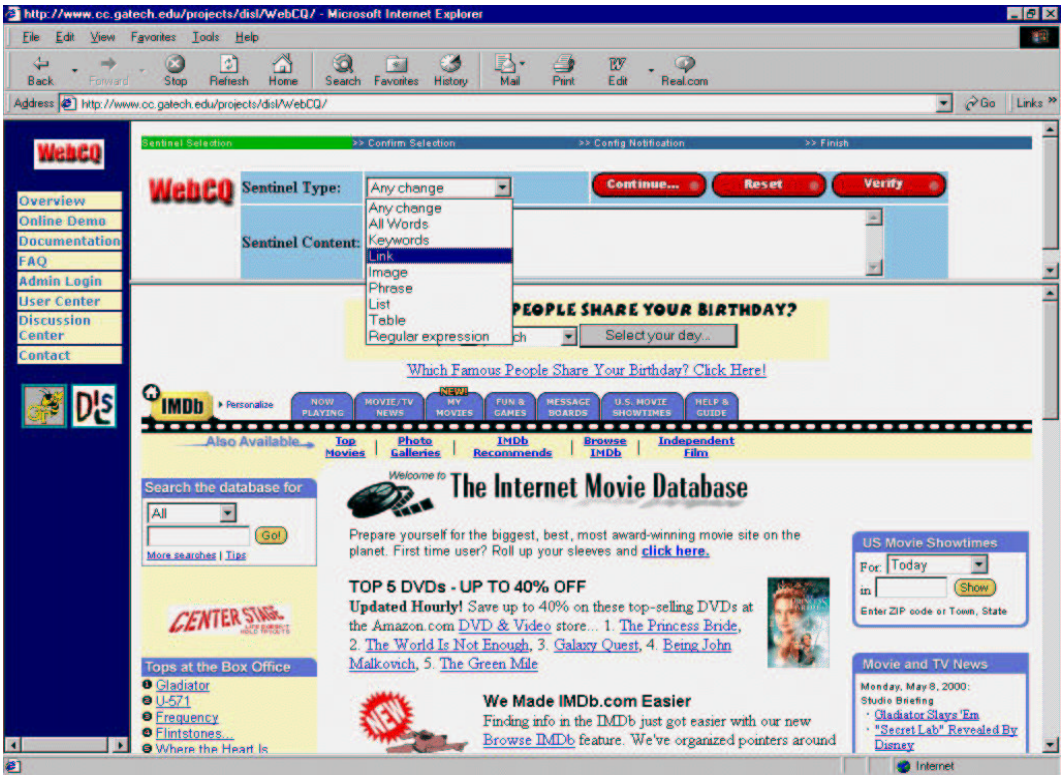

Figure 18: Installing a WebCQ sentinel using the proxy service

Figure 18. We call each registered request a web page sentinel. Sentinels in WebCQ are modelled as continual queries [62] and are persistent objects. The sentinel installation is submitted to the sentinel installation manager once the user has entered her email address and her preferred monitoring frequency, notification frequency, and notification method. The sentinel installation manager first translates the sentinel request into a continual query expression [62], and then registers the continual query in the WebCQ metadata repository. The WebCQ daemon fires the change detection robot periodically to evaluate all installed sentinels. Once interesting changes are detected, the difference generation and summarization module will be fired to generate a detailed change notification report for each sentinel and a change summarization for each client. A notification will be fired to notify users of the interesting changes.

Users may monitor a web page for any change or specific changes in images, links, words, a chosen phrase, a chosen table, or a chosen list in the page. Furthermore, WebCQ allows

users to monitor any fragment in a Web page, which can be specified by a regular expression. Table 3 shows a list of basic sentinel types supported in WebCQ.

Table 3: Basic Sentinel Types in WebCQ

sentinel types	**Synopsis**	**Sentinel Monitoring Method**
Any Change	Any update on the page object	Watch any change to the modification timestamp of the file, compare MD5 hash values
Link Change	Any change to links of the page	When new links added or old links removed
Image Change	Any change to images of the page	When new images added or old images removed
Words Change	Any change to words of the page	When new words added or existing words removed
Phrase Update	Any change to the selected text phrase	Identify and extract the selected phrase; detect any change in the phrase
Table Change	Any change to the content of the table	Identify and extract the selected table; detect any change to the table
List Change	Any change to the content of the list	Identify and extract the selected list; detect any change to the list
Arbitrary Text Change	Any change to the text fragment specified by a regular expression	Identify and detect any change; in the selected fragment
Keywords	The specified keywords appear in or disappear from the page	Detect if the selected keywords disappear or appear

Users may register their update monitoring requests using basic sentinels or composite sentinels. A composite sentinel type is a composition of two or more sentinel types. A sentinel is either an instance of a basic sentinel type or an instance of a composite sentinel type. The operators used for sentinel composition are omitted here due to the space restriction. The syntax of a WebCQ sentinel is partially described as follows (the full specification is included in Appendix B):

```
<WebCQ Sentinel> ::=
   CREATE Sentinel [<sentinel name>] AS
        Sentinel Type: <sentinel type>
        Sentinel Target: <sentinel target>
        Sentinel Object: <object desc>
        Trigger Condition: <time interval>
        Notification Condition: <time interval>
        [Notification Method: <method type><method signature>]
        Start Condition: <time point>
        Stop Condition: <time point>
<sentinel name>::= String
<sentinel type>::= <HTML-specific-type> | <general-type> | <rule-type>
<HTML-specific-type> ::= 'All Links' | 'All Images' | 'Table' | 'List'
<general-type> ::= 'Any Change' | 'Phrase' | 'All Words' | 'Keyword'
<rule-type> ::= <regular-expression>
<sentinel target>::= <URL>
```

```
<object desc>::= String
<time interval>::= Integer {'MINUTE' | 'HOUR' | 'DAY' | 'WEEK'}
<time point>::= <Month>'-'<Day>'-'<Year>' '<Hour>':'<Min>' '<TimeZone>
<method type>::= 'EMAIL' | 'WEB BOARD' | 'LOGFILE' | 'EXTERNAL' | 'FAX' | 'PAGER'
```

The trigger condition of a sentinel specifies how frequent the change detection robot should be fired to check whether any interesting changes have happened. The notification condition specifies the desired frequency for receiving change notification from the WebCQ system, and is designed to support situations where the desired notification condition is different from the trigger condition. It can be the same as the trigger condition or n times of the trigger frequency. For instance, one may want WebCQ to track changes daily but receive the notification report weekly. The optional clause `Notification Method` allows users to select a notification means from the set of methods supported by the system. In the first prototype of WebCQ, we only support two notification methods: email and personalized web bulletin. Due to the space limitation, we omit the complete syntax of the CQ specification language in this paper. Readers who are interested in more details may refer to [64]. A WebCQ sentinel example is given by Example 3.

Example 3 *Consider a WebCQ sentinel "track any change on the 'IMDb Movie of the Day' from the front page of the Internet Movie Database Web site". The installation the through WebCQ GUI is shown partially in Figure 20. The user may select regular expression as the sentinel type in the monitoring condition, namely "*`IMDb Movie of the Day .*?more.*?\)`*". WebCQ will automatically highlight the paragraph on IMDb Movie of the Day and capture and store this sentinel in the following WebCQ sentinel expression (in the form of a continual query [60, 62]):*

```
CREATE Sentinel movie_of_the_day AS
  Sentinel Name:          "IMDB movie of the day"
  Sentinel Type:          regular expression
  Sentinel Target:        http://www.imdb.com
  Sentinel Object:        "IMDB Movie of the Day.*?more.*?\)"
  Trigger Condition:      1 day
```

```
Notification Condition: 1 day
Notification Method:    EMAIL (john.doe@somedomain.com)
Start Condition:        November 1, 2003
Stop Condition:         October 31, 2004
```

The default duration for a continual query (CQ) is one year starting from the time of the installation. Our experience with the continual query system [62, 68] shows that the capability of supporting both trigger condition and notification condition is especially useful for the situations where a user prefers to receive a summarization of all the change detection reports of a sentinel periodically rather than receiving an email whenever the sentinel is evaluated and changes are found. Such sentinel-specific summarization is also useful when notification of changes to web pages is to be sent to an application program to trigger an action, rather than or in addition to sending it by email to a user. In WebCQ, the trigger condition and the notification condition are set to be the same by default. The notification model of the WebCQ system will be discussed further in Section 3.8. In the second release of the WebCQ system, we also allow triggers to be content-sensitive. For instance, users can specify a monitoring requesting for tracking changes on the Movie of the Day news from the Internet Movie database front page, with a trigger condition that requires to deliver the change only when more than 50% of the content in the Movie of the Day news has changed. Another example is *"notify me when Panic Room starring Jody Foster becomes the movie of the day"*.

Other important features of the WebCQ architecture include its server-based proxy cache service. The goal of introducing a proxy cache service is to reduce the cost of repeated connections to the remote information servers. The proxy service works as follows: Any remote page request will be sent to the nearest proxy server first. The proxy server only forwards the remote fetch request (such as HTTP Get or HTTP Post) to the corresponding information server if a copy of the requested page is not found in the local proxy cache or any of the relevant proxies. The static page crawler is used to fetch static pages, while the dynamic page crawler is designed for handling the remote fetch of dynamic pages, including search interface extraction and HTTP query composition. An obvious advantage of the

proxy service is to enable the WebCQ system to share the cost of remote page fetch among all monitoring requests over the same web page. By reducing the frequency and the overhead of network connections to remote data servers, the unnecessary network connection cost is avoided, and the scalability of the system is enhanced.

In the subsequent sections we focus our discussions on the following three main components of the WebCQ system: change detection, difference generation and summarization, and change notification service.

3.4 Change Detection

Change detection is crucial to change monitoring services. The main tasks of change detection include accurately extracting and identifying objects on the target Web page and detecting changes between the page's last captured snapshot and the current copy.

In WebCQ, the change detection robot consists of three modules: object extraction, sentinel evaluation, and object cache update. For each sentinel, the change detection robot first fetches the web page being monitored by the sentinel through a page fetch call to the proxy server (recall Figure 17). Then it performs the following three tasks: *object extraction, sentinel evaluation, object cache update*, which will be discussed in the following sections.

3.4.1 Object Extraction

In order to detect changes on a Web page, we have to first identify the objects residing on the page. The *Object Extraction* task locates and extracts the objects being monitored from the page. In WebCQ, we provide three different methods to extract web page contents of interest. They are HTML-based data extraction, rule-based (using regular-expression) data extraction, and general syntax-based data extraction, as shown in Figure 19.

- For HTML-specific data extractions, four different logical structures are extracted, including *Table, List, AllLink and AllImage*. A small token-based HTML parser is used to identify HTML tags and extract logical structures of interest from a given Web page.

- For rule-based data extraction, WebCQ identifies and extracts any text string specified

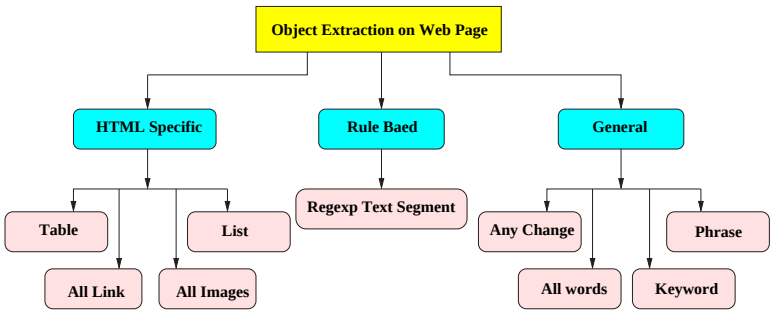

Figure 19: Object Extraction in WebCQ

by a Perl-like regular expression[1]. For instance, the regular expression "`<td><b>(Java.*?)</b></td>`" extracts a table cell containing a string starting with 'Java' in bold font. Recall Example 3 in Section 3.3, it is a regular-expression sentinel, which tracks changes to the "IMDB Movie of the Day" information on the front page of Internet Movie Database web site. The target object being monitored is shown in Figure 20 as the shaded area in the lower frame as well as in the pop-up window. The *Regular Expression* sentinel type is an important feature of WebCQ. An arbitrary text object in a Web page can be constructed using a *regular expression*. Almost all other data object types can be transformed into a regular expression type. It is well known that different regular expressions have different evaluation cost. In the current prototype of WebCQ system [109], we use the regular expression package from GNU (gnu.regexp). One of our future research efforts is to support the transformation of user-defined regular expressions into equivalent but more efficient ones.

- For general syntax-based data extraction, in addition to sentinels of any change type, three types of logical objects are extracted: *Keyword, Phrase,* and *All words*. Standard Java String and Tokenizer packages are used in the WebCQ implementation. For any change sentinels, WebCQ first uses the last modification time stamp to determine if the page has changed at all. If yes, WebCQ will run a comparison of the MD5 [86] signature of the new and old Web page being monitored before computing the actual differences.

[1]See http://www.perldoc.com/perl5.6/pod/perlre.html for details about Perl regular expressions

Figure 20: Object extraction during a sentinel installation

To make the paper self-contained, a brief description of the basic concepts used in this section is given in Appendix C.

3.4.2 Sentinel Evaluation

The Sentinel Evaluation module compares the object extracted from the current copy of the page with the previous cached copy of the object. The difference between the two copies of the object is computed and changes are detected. The sentinel change detection algorithm is described in pseudo code in Appendix D. Here we give an informal overview of the sentinel evaluation process.

- For detecting "any change" to a page, we use a set of progressive techniques as described below:

  First, we use the HTTP `HEAD` request to obtain the last-modified timestamp for all static pages. Occasionally, the last modification timestamp alone may not be enough.

For example, a page whose last modification timestamp changes but no changes actually occur in its body may be flagged as changed. One may use a `HEAD` request combined with a checksum evaluation to handle such situations.

In fact, several methods can be used in addition to the last modified timestamp to double-check if a page has changed. For example, once a `HEAD` request indicates a change, a simple solution is to use the file size in addition to the modification timestamp. This method will avoid those cases where the content of a page did not change though the timestamp is changed. However, it will miss the cases where a page has changed but the amount of text deleted equals the amount of changes inserted, leaving the file size unchanged. From HTTP 1.1 on, a "Content-MD5" header field is provided[47] as an MD5 digest (128 bits) of the document body. If this field is present, it can be used to compare with the previous stored MD5 digest of the page to detect if change happens to the document. However, at this point, we do not know what has been changed.

A better approach is to use `HTTP GET` to retrieve the page and compute the difference. Obviously, generating the difference will give a more accurate detection result but it is also more expensive. In the situation where no last modification timestamp is provided such as the case of dynamic pages or a prior attempt that no timestamp is given, then HTTP `GET` is used to retrieve the body of the page via the proxy service and a checksum (such as MD5 digest) is computed.

- To detect changes to hypertext links or images in a page, the hypertext reference tag (e.g., `<A href=...>`) is used to identify all hypertext links in the page; and the image tag (`<IMG src=...>`) is used to identify all images in the page. For *All links, All images* and *All words*, they are all set-based sentinel types. We detect object insertions and deletions using set difference operations.

- For sentinels of types *Phrase, Table, List*, the task of identifying and extracting the correct object begin monitored in a page is more complicated. We introduce the concept of context bounding box. A content bounding box for an object being monitored

in a web page is defined by the surrounding context of the object. This context-bounding box will be used to identify and extract the object in the subsequent copies of the page. There are several ways to define the bounding box context for an object. For example, we may define the bounding box context of an object by a given number of words before and after the boundary of the object. The context bounding box approach assumes that the selected text surrounding the object being monitored is relatively stable.

The context-bounding box is particularly useful for handling duplicates of objects in a page, i.e., the objects that have the same contents within a page. It is understood that the chosen bounding box should be more stable than the object being monitored to ensure the precision of the object location and extraction. The quality of the monitoring to some extent depends on how we choose the bounding box for a given sentinel and whether and how often it may change. In the first prototype of WebCQ, we are using ten words before and ten words after the object to be extracted to define its bounding box. However, the selection criteria for bounding-boxes should be a configurable parameter. One can adjust it to use different selection criteria for defining the bounding box for different types of Web pages.

Figure 21 sketches the context-bounding box, and a set of examples, including a case where the bounding box is stable and representative cases in which the bounding box can be changed. We use B to denote the beginning of the bounding box for the object O and E the ending bounding box for O. An updated version of the object being monitored is in shaded area. We represent changes to the bounding boxes (either beginning or ending) as corner wedges.

Case (1) in Figure 21 shows the stable situation where the object being monitored is changed and the bounding box is not. The sentinel change detection algorithm works the best in this type of situations. Cases (2), (3) and (4) are not considered as a change since there is no change to the object of interest. For changes similar to those shown in Cases (5), (6), and (7), we need to dynamically adjust the bounding box to

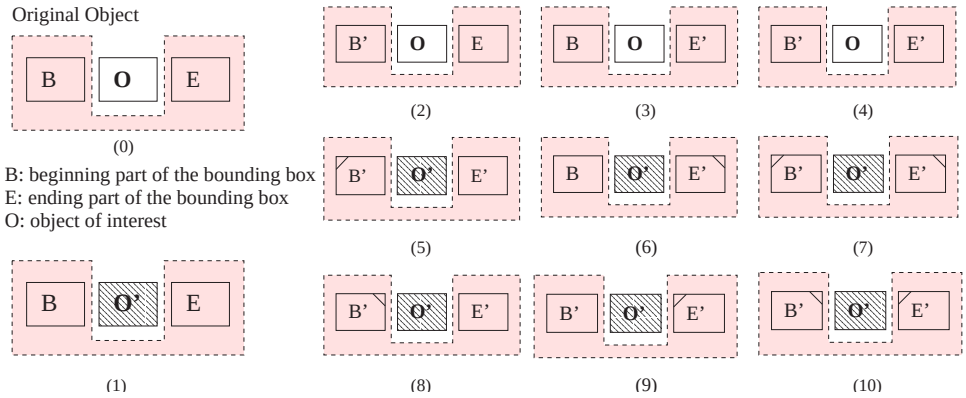

Figure 21: Context bounding box and types of changes

precisely locate the object. In the first prototype of WebCQ, if less than 50% of the words used in a bounding box have changed, then we adjust the bounding box to the unchanged portion. For example, let B = $< w_1, w_2, ..., w_n >$ (n=10 by default), and $B^{'} = < w_1^{'}, w_2^{'}, ..., w_i^{'}, w_{i+1}, ..., w_n >$, i>1, be the changed version of B. That is, only the first i words in the beginning bounding box B has changed, leaving the part closer to the object being monitored $O^{'}$ intact. We can adjust the beginning bounding box to be of length i instead of 10 if $i < 5$ holds. Similarly, let E = $< w_1, w_2, ..., w_n >$, and $E^{'} = < w_1, ..., w_i, w_{i+1}^{'}, ..., w_n^{'} >$, $1 \leq i < n$, be the changed version of E. Only the last $n - i$ words in the ending bounding box E has changed, leaving the part closer to the object being monitored $O^{'}$ intact. We can adjust the ending bounding box to be of length i instead of 10 when the amount of changes is less than 50%, i.e., $i < 5$. A problem may arise when both beginning and ending boxes B($B^{'}$) or E($E^{'}$) are substrings of the object $O^{'}$, or vice versa. Therefore, in the current version of WebCQ, we do not let users monitor an object that is too small (e.g., less than 5 words for the *Phrase* sentinels).

Recall again Example 3 in Section 3.3, instead of using regular expression sentinel, an alternative way is to express the same tracking request using the phrase sentinel and set B=<"IMDb","movie","of", "the","day"> and E=<"more",")"> to identify the paragraph under "IMDb Movie of the day" as the phrase object to be monitored. This example will generate the same sentinel object as the one shown in Figure 20.

Finally, the algorithm will fail for the last three cases (i.e., (8), (9) and (10)) in which either beginning or ending bounding box is changed in the portion of the box that are close to the object being monitored.

Our experience has shown that for static web pages, using words instead of tags to define the bounding box context of an object often gives us more accurate results. However, for dynamic pages, using tags instead of words tends to provide results of higher accuracy. For a table or a list object, users may select a sentence or a keyword within the table or the list as the unique identifying text of the table or list. This mechanism also applies to phrase objects. Whenever users provide such identifying text, instead of using the bounding box context method, WebCQ will then use such identifying text to identify the object.

- For sentinels of type *Regular expression*, the object content is represented by a Perl-like regular expression. To detect a change in a Web page, we simply compare the new regular expression evaluation with the cached evaluation copy. The comparison is based on character string match.

3.4.3 Object Cache Update

We have discussed the first two steps in the change detection process of the WebCQ sentinels. Object Cache Update module is the last step. It is fired if a change is detected for a particular sentinel upon the completion of a sentinel evaluation.

More concretely, given a page sentinel, the change detection robot first fires the object extraction module that performs two tasks: (1) It fetches the current copy of the page through the proxy service. (2) Then it uses the bounding box of the sentinel object to locate and extract the phrase in the current copy of the page. Then the sentinel evaluation module is fired. It loads the cached copy of the sentinel object, which was extracted from the previous copy of the page, and compares the cached copy with the current copy of the sentinel object. If no difference is found, the change detection ends without taking any action for summarization or notification. An evaluation log is recorded. If a difference is found, the following three actions are performed:

- The change summary generation component is triggered first. The difference is computed between the two copies of **O**. A detailed change report is created and the personalized change summarization is updated.

- The object cache update module is fired to refresh the cache copy of the object **O** to the current (new) copy O'.

- The notification manager sends an email notification to the corresponding user with a brief summary of the changes and a link to the web page where a personalized change summarization report can be found.

3.5 Change Semantics and Relevance

Once we have all the basic elements implemented by the *Change Detection Robot*, the next relevant question is: what are the changes interesting to our business processes? Successfully answering this question is essential to integrate a Web page change monitoring service with business processes. The first step in providing a mechanism to create an intelligent response to a change is to evaluate the semantics of the recognized change.

The basic sentinel types in WebCQ are described in Table 3. These sentinels are created to track relevant page changes, such as hyperlinks, images, words, phrases, lists, and tables. Some changes are document specific (e.g. links, images, list, and tables). Others are not (e.g. words). Initial determination of page properties will be fed to the change response engine. These basic types of sentinels are the building blocks for identifying application domain specific changes.

Many of the documents accessible on the Web are semi-structured (HTML, XML, etc.). The documents contain a certain level of semantics in the markup itself. This combined with the actual value of the change can at times be a good determiner of the semantics of the change. There are several classes of changes:

1. Structural changes - those that change the location of information within a document (e.g. moving the address field from the header to the footer) or relevant location within a collection of documents (e.g. structure change of the hyperlink tree where

the target document resides). This type of change is generally qualitative.

2. Informational changes - those that change the actual information within the document (e.g. change the street name in the address field) that do not need further interpretation. This type of change is quantitative.

3. Presentational changes - those pertinent to visual changes (e.g. font, color, style changes in the document). This type of change is qualitative.

4. Semantic changes - those that impact the semantics of the information (e.g. moving the street address from primary address to secondary address). These changes are application-dependent. They can be both qualitative and quantitative changes.

These classes of changes provide the first guidance for users to determine change relevant to business needs. The most important changes are semantic changes, which are generally the most difficult to model and capture. However, with the help of domain experts or automatic and semi-automatic ontological analysis tools, the problems can be reduced.

Relevance in this context is meant to be the relative "worth" of a change. This value will then provide hints to subsequent systems on how they should react to the change. If the relevance is low, perhaps the information is not forwarded to the change response system, but is instead only used to notify appropriate people of the change. In all cases, when an automated response is taken, a notification is generated to alert the appropriate people of the change and its response.

Change relevance is a difficult parameter to measure. There are many variables involved in determining the value to assign any particular change. The first steps being taken by the authors is to begin to categorize the types of changes and begin to catalog the response generated based on the type and value of the change. This information is then fed back into the system to improve its accuracy and predictability. This new information is then used to provide hints to the user based on previous experience.

Integrating a change monitoring service with business processes and incorporating change semantics and relevance with specific application domains are beyond the scope of this thesis. Interested readers can refer to [101] for more details.

3.6 Difference Generation and Summarization

Most of the tools that monitor changes to web pages have the capability of notifying users that something on a page has changed with the link to the new copy of the page. But few are able to include what and how the page has changed in the notification report. When the number of pages that a user is interested in tracking changes is large and the changes on the pages are subtle, it is likely that the user will not know what has changed by simply viewing the new copy of the page and comparing it with what the user has remembered when the page was last seen. Therefore, computing and showing the difference to a web page is a critical component of an information monitoring system from the usability perspective. In this section we address three issues related to representing and viewing changes: difference generation, difference representation, and change summarization.

3.6.1 Difference Generation

Computing changes to a web page consists of two tasks: obtaining both the old and the new version of the page and comparing the two versions to display the difference.

The first task is related to the archival of web pages. Most of content providers only provide access to the most current version of their web documents. Although some content providers may maintain a history of their documents, for instance using the Revision Control System (RCS), to our knowledge few provide world-wide access to their archives. Furthermore, archiving every page accessed demands careful consideration on technical issues such as storage, indexing structure, retrieval efficiency and legal issues such as copyright protection on archival [30]. In WebCQ, we deliberately choose not to archive every page we access. Instead, for every web page accessed by WebCQ we at most keep one past copy that will be used to compare with the current version of the page and compute how changes have been made. Saving an old version of a page is not significantly different from a proxy cache server keeping a copy of a page until the page reaches its expiration timestamp. In contrast, the object cache update module discussed earlier is designed for maintaining and refreshing the object cache used by both the sentinel evaluation module and the difference generation module (recall Figure 17).

The second task is related to the difference generation from two versions of a web page. One way to compute the difference between two versions of an HTML page is to use *HTMLDiff* developed by AT&T [30, 29]. Similar to the UNIX *diff* utility [51], *HTMLDiff* takes two versions of a page as an input and produces a new and merged HTML document. The new document highlights the differences between the two versions by flagging the inserted text with bold or colored face and deleted text with a horizontal line cross over. Changes to existing text are treated as deletions followed by insertions. As pointed out by Fred Douglis and his colleagues [30], every invocation of the *HTMLdiff* may potentially consume significant computation and memory resources. Such resource overheads will restrain the number of difference operations a server can perform at a time, limiting the scalability of the system. In WebCQ, most of the sentinels, except *any change* sentinels, are targeted at tracking changes to a page fragment rather than the entire page. A page fragment can be a specific HTML object (such as phrase, list, and table), an arbitrary text fragment, or a specific component of the page (such as links, images, and words). In such cases, a simplified difference generation algorithm should be used to reduce the overhead of the general *HTMLDiff*.

There are at least three basic mechanisms for computing the difference between two versions of an HTML fragment.

- **Difference is flagged only when content (raw text in between a pair of tags) changes.**

  This method views an HTML fragment simply as a sequence of words. Markups and extra whitespaces are ignored for the purpose of comparison. Therefore when a text paragraph comprised of five sentences is changed into a list of five items (each sentence starting with a <LI>), no difference is flagged because the content of the paragraph version matches exactly the content of the list version, although the presentation structure has changed.

- **Difference is flagged when either content or structure changes.**

  This method views an HTML fragment as a syntax parse tree with tags as internal

nodes and raw text as leaf nodes. The subtree-equality comparison algorithm is used to compute the difference between two versions of a fragment. Obviously, in this case a subtree representing a paragraph of five sentences will be different from a subtree representing a list of five sentences. Thus, the example of changing a paragraph with a list will be flagged, even though it was merely a format change.

- **Different flags are used for content changes and structure changes.**
  This method can be implemented by applying both methods above. The application of the first method detects if the content has changed. The application of the second method tells if any format change has occurred. Thus, for the example of changing a paragraph with a list, the comparison will display no change to content but a change to formatting.

One problem with the second and the third method is that any small and trivial change to formatting, such as adding or removing a line-break tag `<br>` in the middle of a sentence or adding a paragraph begin tag `<P>` in between two sentences, will be flagged. After careful consideration, the first mechanism is used, in the current prototype of WebCQ, for computing the difference from two versions of an HTML page fragment.

An ongoing research effort is to design a structure-aware change detection and difference generation algorithm, called Sdiff. The technical detail and initial experimental results on Sdiff can be found in [85].

3.6.2 Difference Presentation

There are three popular ways to present the difference between two web documents [30, 78, 76]. The first approach is to *merge the two documents* by summarizing all of the common, inserted, and deleted contents in one document, as is done in *HTMLDiff* [30] and Microsoft Word product. The main advantage of this approach is that all the changes are embedded in one document with the common unchanged part of the two documents displayed only once. However, showing all the change information in one document may make it difficult to read especially when the documents are large or there are many differences between the two.

Figure 22: A difference presentation for "Any Change" sentinel on CNN.com

The second approach is to *display only the differences* and omit the common parts of the two documents. The GNU Diff utility [37] falls in this category. This approach is beneficial for large documents or two documents with much in common. The drawback is that the change context is lost. For Web pages, this can also lead to confusing presentations.

The third approach is a *side-by-side presentation* of the differences between two documents. This method enables uses to view bi-directional changes to both the old and new documents. Although this method also has the problem of presenting difference for very large documents and documents with a lot of changes, the side-by-side presentation is most intuitive for visually comparing documents by humans. There are also options for side-by-side presentations. For example, we can choose to display either the old document or content with deleted parts marked.

In WebCQ we use a hybrid approach that utilizes all three of the presentation schemes in selected combinations. For instance, we use the *Side-by-Side* for sentinels of phrase or regular expression type, and the combination of *Side-by-Side* with *Only Difference* to

display differences for all-links, all-words, table, and list sentinels. We use the merge-two-documents approach for sentinels of image type. Figure 22 shows the difference presentation for an "Any Change" sentinel installed on CNN.com home page. The difference presentation module displays the new web page with the new text highlighted on the right window and the old copy of the web page with the deleted text highlighted in a different color on the left window. Other example screenshots on difference presentation are available online at the WebCQ project page [109].

Besides the three popular ways for presenting differences between documents, there are also approaches that present page changes through analysis of the pages' hyperlink structures. We can use a tree visualization to illustrate the relative changes. When in a closed enterprise environment, changes to other documents that have links within the target page often means relevant changes to the target page as well (e.g. a general guideline page has a link to a detailed manual document). Figure 24 is an example of difference presentation in a hyperbolic tree view [2]. The green nodes indicate newly available documents. Green edges indicate newly added hyperlinks to the target document D(node marked with light blue color). The corresponding *side-by-side* difference view for the target page D is shown in Figure 23.

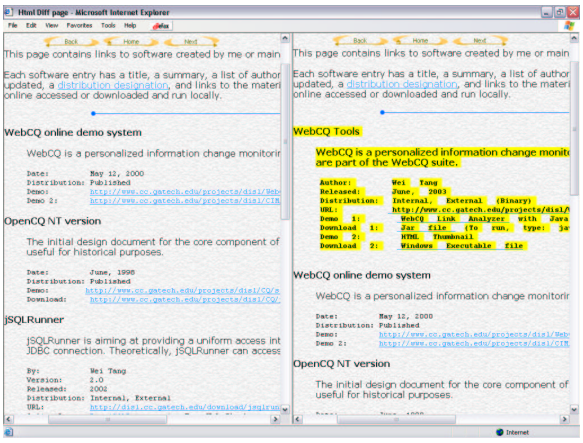

Figure 23: Difference Presentation with Side-by-Side View

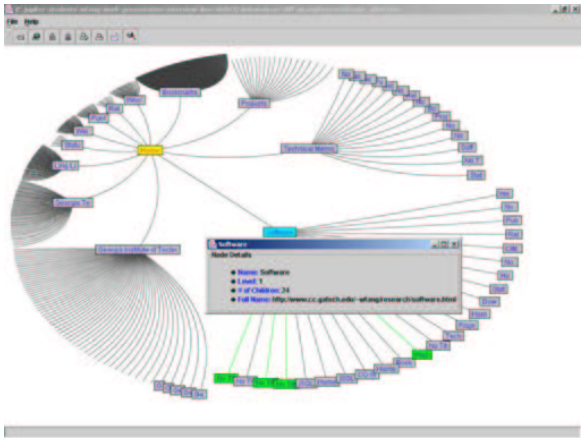

Figure 24: Difference Presentation in a Hyperbolic Tree View

The tree view works best when the page has little presentational change whereas the

[2]The hyperbolic tree view is generated by the *WebCQ LinkAnalyzer* tool [109]

underlying links contained in the page changed. The tree visualization approach also works well for site-level change detection and among a set of relevant documents to depict the change relationships.

3.6.3 Change Summarization

There are two types of summarization: *per sentinel summarization* and *per user summarization*. Figure 22 of previous section demonstrates the *per sentinel summarization*: in addition to the presentation of how the page has changed (i.e., the difference presentation), every detailed sentinel evaluation report in WebCQ also includes a brief summary of the sentinel, such as the sentinel type, the sentinel content, the title and the URL of the web page monitored by the sentinel. A brief change statistics of the particular page is also displayed in the lower frames about the total number of object changed (deleted and inserted) and number of links changed (deleted and inserted). Such summarization gives users a quick hint on how much the page has changed quantitatively (besides visually).

Our experience with the change monitoring systems shows that one of the important usage improvement for change tracking and notification services on the Web is to provide at least one change summarization for each user, reporting the status of all sentinels installed by the user (i.e., *per user summarization*). Users will be given a choice of receiving a summarization notification for all sentinels they have registered with the system or one notification per sentinel. A change summarization report contains a list of summarization records. Each record presents a brief summary of the recent sentinel evaluation result, including:

- the URL of the web page being monitored,
- the type of change, such as insertion, deletion, update to the page, a new page, or a deleted page,
- the sentinel type,
- the last modification timestamp, showing when the page has changed,
- the last evaluation timestamp, displaying when the change is detected,

- the last notification timestamp, showing when the last notification was sent out,
- the timestamp when the page was last seen,
- the installation timestamp, showing when the sentinel was registered initially,
- the termination timestamp, showing when the monitoring sentinel will be terminated,
- the number of notifications since the installation,
- the number of evaluations since the installation, and
- the hypertext link to the detailed difference presentations of the sentinel.

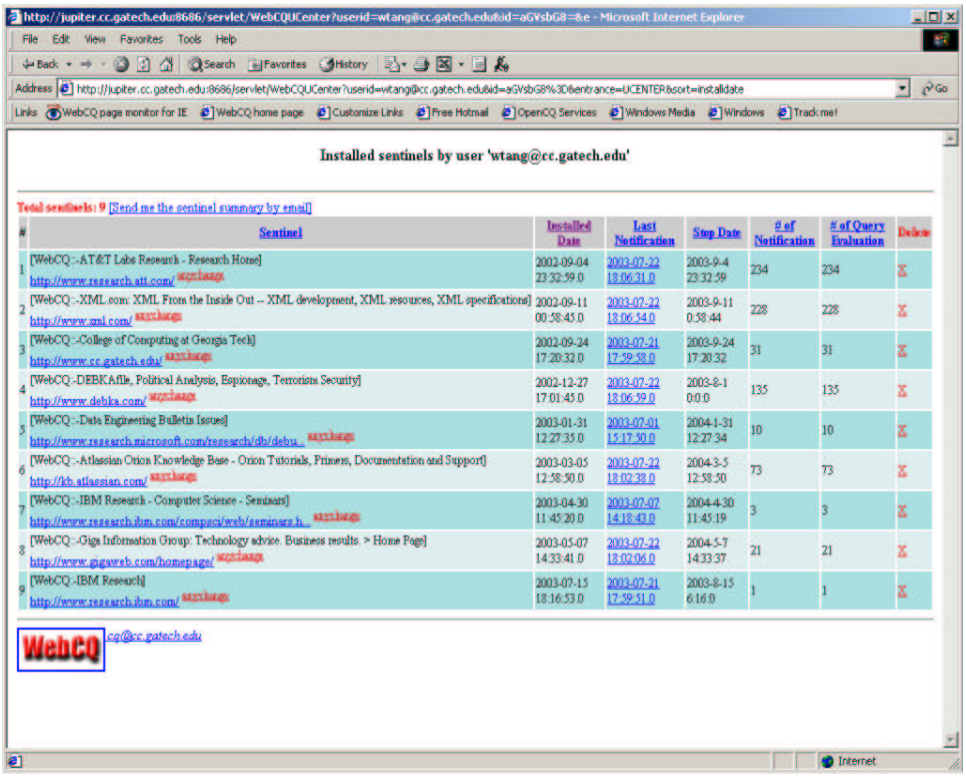

Figure 25: A *Per User* Change Summarization Example

Most of the information above is recorded and maintained by WebCQ during the sentinel processing. For all static pages, the HTTP `HEAD` request can be used to obtain the last-modified timestamp. However, if WebCQ has knowledge from a prior attempt that no timestamp is given, then HTTP `GET` is used to retrieve the body of the page and compute a simple checksum. Figure 25 shows an example of *per user* change summarization.

Change summarization listing all changed pages (possibly classified in categories) is particularly important when a user wants to track a large number of web pages and installs tens or hundreds of sentinels. This is simply because using individual email messages, each corresponding to one sentinel, would easily overwhelm the user. Such situations may be aggravated when multiple updates to a page were reported in a sequence of email notifications before a user could process the mails.

3.7 Proxy Cache Service

A proxy server offers a web caching solution. Instead of caching popular web pages on clients' machines, a proxy server caches popular web pages close to the information server where such web pages are requested frequently and may be used for different processing purpose. To avoid remotely fetching the same web page many times within a short time span, a proxy caches the most frequently asked web documents in the proxy cache. Thereby, a popular document requested by clients, if can be found in the proxy cache, will result in a hit, which not only saves network bandwidth but also lowers the access latency for such documents.

In this section we review general issues in the design of a proxy service and discuss the design decisions made in implementing a proxy service for the first prototype of WebCQ.

3.7.1 Proxy Architectures

There are two classes of architectural designs for proxy services: single-proxy and *multi-proxy*. In a *single-proxy* architecture, the proxy acts like a central directory. Every page fetch request is first sent to the proxy. If the page misses in the proxy cache, the proxy will redirect the page request to the appropriate information provider on a remote server. The simple and straightforward implementation is one of the main advantages of the single-proxy architecture. The disadvantage of the single-proxy architecture is also obvious. A single proxy may become a bottleneck (or hot spot), and thereby degrades the performance of a portion of the network or slow down the entire monitoring service system. One of the important design choices is to set a limit to the number of clients it can serve in order to guarantee that using the proxy is more efficient than, or at least as efficient as, direct

contact with the remote servers.

A *multi-proxy* architecture provides an infrastructure that assists proxies cooperate efficiently with each other with an ultimate goal − to increase the probability to hit a document. The *hierarchical-proxy* architecture and the *distributed-proxy* architecture are the two popular *multi-proxy* architectures. The hierarchical proxy cache cooperation was first proposed in the Harvest project [22].

In a *hierarchical-proxy* architecture, caches are placed at multiple levels of the network. When a requested page is not found in a proxy's local cache, the request will be forwarded to its parent proxy. This process continues recursively. If the proxy at the root of the hierarchy cannot satisfy the request, the root proxy will redirect the page request to the original information provider at the remote server. When the requested page is found, either at the cache or at the original remote server, it travels down the hierarchy to the proxy where the request was initially received from a client, leaving a copy at each of the intermediate proxy caches. Hierarchical proxy architecture is more bandwidth efficient, especially when some proxy servers do not have high-speed connectivity. Also popular web pages are provided with fast access, as they are efficiently diffused towards the demand. However, as pointed out in [103], one problem with the hierarchical proxy architecture is that it requires a significant coordination among participated proxy servers to set up such a hierarchy that the proxy servers are placed at the key access points in the network. The second problem is the high storage cost since many copies are stored at different cache levels. Also higher level caches may become bottlenecks and have a long queuing delay.

In the *distributed proxy cache architecture* [84, 103], each request is sent to the nearest proxy. Multiple proxy servers are organized in a network instead of a hierarchy. They cooperate together to serve each other's misses. The data pages are not replicated but each proxy keeps a local copy of the information about the location of the documents. Summary cache [32] is one example implementation of the distributed proxy cache architecture. The efficiency of proxy co-operations is an important issue in this scheme. Mechanisms are needed to limit the cooperation between neighbor proxy caches to avoid obtaining documents from distant or slower proxy caches when they can be retrieved directly from the

original server at a lower cost.

For simplicity, The single-proxy architecture is implemented in the first prototype of WebCQ with the capability to replicate the proxy when the number of requests exceed certain thresholds. One of the extensions to WebCQ is to incorporate a distributed proxy cache in the next generation of WebCQ.

3.7.2 Cache Replacement Policies

Cache replacement is one of the key aspects for the effectiveness of a proxy service. An efficient cache placement/replacement algorithm is the one that can yield high hit rate while minimizing the average latency and the total cost.

The traditional and most commonly used replacement policies are the *Least Recently Used* (LRU) algorithm, which evicts the document which was requested the least recently, and the *Least Frequently Used* (LFU) algorithm, which evicts the document which is accessed least frequently. A number of cache replacement algorithms have been proposed, which extend LRU and LFU policies. Instead of evicting documents that are least recently used or least frequently used, one type of extensions is to evict documents based on the lowest download latency [114], the largest document size [116], the least recently used documents whose sizes are above a given threshold. Another type of extensions is cost-based replacement policies. Algorithms in this category use cost functions or utility functions to evict documents with the lowest cost or the least utility.

In the proxy service of WebCQ, the LRU algorithm is used for cache replacement.

3.7.3 Cache Coherence Mechanisms

Using proxy and proxy cache reduces the access latency with a price − pages in the cache may be out of date with respect to their master copies on the remote web servers from which the pages are originated. Cache coherence mechanisms define when and which documents need to be refreshed.

In WebCQ, a periodical refresh mechanism is used to maintain the cache consistency. This is because each sentinel has a user-defined trigger condition, which can be viewed as the degree of staleness the users are willing to tolerate. Thus we set the concrete expiration

period for a web page to be the minimum interval of all the trigger conditions used in the sentinels that track changes over the given web page. The WebCQ proxy service uses the conditional GET − an HTTP GET combined with the header *IF-Modified-Since:*`date` to request a remote server to return a copy only if it has been modified since the given date. Another frequently used header is *Last Modified Date*. With this header, every HTTP GET message indicates the last time the page was modified. The header *Pragma:no-cache* is sometimes used to appended to GET to indicate that the page to be reloaded from the server no matter whether it has been modified or not. The reload button offered by most browsers like Netscape uses this header to retrieve the current copy of the page. One can also use HTTP `HEAD` request to obtain the last modified timestamp.

Interesting to note is that although the problem of cache coherence on the Web appears similar to the problem of cache coherence in distributed file systems, not all solutions developed in distributed file systems can be directly applied. This is in part due to the unique characteristics of the Web in access patterns, the large scale, and the single point of updates for web pages [28, 39].

3.8 Change Notification Service

Change notification is a software facility that provides mechanisms for notification of information changes. Many notification services that provide notification of changes to Web pages have a few features in common and vary in several other aspects.

The functionality of a notification service is divided into three components: *what to notify* (what types of information should be delivered for notification), *when to notify* (when users should be notified of changes to the web pages they track), and *how to notify* (how timely delivery of notification can be guaranteed). Beyond this common underlying framework, most systems diverge, from both the architectural design to the technical solutions used to address these three basic questions.

In the rest of the section we review a list of important issues in designing and engineering a notification service and discuss the design decision and the mechanisms used in WebCQ notification service.

3.8.1 Reference Architecture

One of the design goals of a notification service is to make it reusable and configurable, so it can easily be incorporated into other system architectures.

In designing a notification service, the first design choice is to decide whether to build the notification service as a standalone service or as a component of the WebCQ information monitoring and tracking system. In the standalone architecture, the main components are comprised of information source, source change event observer, notification manager, and wrapper set. The wrapper set is bundled with the notification service. This architecture is considered "loosely coupled" with the source change event observer. The observer may be provided by, or combined with, the source and connect to the notification manager using the HTTP protocol. The notification service can also be designed as a component of a system. In this case, the notification service may cooperate with other system components in a tightly coupled fashion. The source observer is usually provided by the tracking system using a set of source wrappers or data extractors. For example, in WebCQ we choose to implement the notification service as a component of WebCQ so that the trigger facility can be reused to implement a richer set of notification conditions such as "every 5 times when the trigger condition becomes true". The WebCQ notification service can reside on the same server as the WebCQ engine or reside on a machine connected to the WebCQ server within a local area network. Compared with the "loosely coupled" notification service, a "tightly coupled" one has the advantage of less network communication overhead, but it loses some flexibility and harder to be implemented generically.

The main task for a notification service that cooperates closely with a change detection robot is to synchronize with the robot when changes to web pages are observed, and provide efficient notification of changes to appropriate users or user community. In addition, the notification service may enforce application-specific delivery constraints, such as delivery based on specific user-defined priorities, access controls, and security constraints.

3.8.2 Framework Design: Building Blocks

The basic building blocks for a notification service are the suite of mechanisms that identify when a notification should be sent, how a notification is sent, and what should be included in a notification.

When to Notify –

Eager notification, periodic notification, and on-demand notification are the three mechanisms we have investigated.

- *Eager notification* advocates that any interesting change once detected should be delivered to the client immediately. An obvious advantage of eager notification is to guarantee the minimum latency from the time changes are detected to the time the notification is sent out. Periodic notification and on-demand notification, on the contrary, are based on a deferred notification scheme.
- *Periodic notification* for changes have the advantage of ensuring that the user will have relatively up-to-date information about the pages being tracked, with a bounded freshness value. However, periodic notification requires network connectivity at the time the notification is sent out and for the duration of the update notification which is proportional to the number of pages to be monitored.
- *On-demand notification* promotes the idea of having users fire the notification request when they want to review the change tracking results. On-demand notification will not be effective if the number of web pages being monitored is large, as the user may become overwhelmed with a large number of changes.

In WebCQ, the notification service runs on the server side, thus on-demand approach is not considered. We choose the periodic notification over the eager notification because immediate notification may not scale well when a user registers a large number of sentinels for tracking changes to the web pages. The periodic interval is defined by the notification condition given by the user at the sentinel installation time.

How to Notify –

There are three issues involved with how to notify users of changes: notification initiation, notification mechanism, and change presentation.

- *How to initiate a notification*

  In WebCQ, the notification can be provided by either a server-initiated push delivery or a client initiated pull delivery.

- *How to carry out a notification*

  In the current WebCQ system, both email and web pages are used as mechanisms for server-pushed delivery. For client-pulled delivery, WebCQ currently only provides web pages with a search interface to allow users to query and view the change reports from anywhere in an ad-hoc style. Other means of notification include pager, fax, cell phone, or PDA. More notification methods will be provided for the WebCQ service.

- *How to present changes to users*

  Prioritization and summarization are the two major mechanisms we have investigated for ensuring an effective and pleasant presentation of change notification to the users. Prioritization allows the notification service to bring web pages of particular importance to the attention of a user at the first glance. Summarization allows the users to see an overview briefing of the current status of all web pages to which they track changes before going to the detailed change report of each sentinel.

We have investigated three different mechanisms for prioritizing a list of notifications on a per-user basis. The first one is to order a list of sentinels in a change summary report in a reverse chronological order of modification dates, with the most recently modified page on the top of the report. The second mechanism is to let users assign a priority number for each sentinel they install and then use the user-defined priority numbers to determine the order of sentinels in the change summarization report. The

third approach is to let users prioritize the topics of interest and within a topic the web pages are organized in a reverse chronological order of modification dates.

Often users wish to know the amount of changes to the web pages before visiting the detailed difference presentation document. Examples of such types of summarization include the type(s) of changes found, the number of links added or deleted, the fraction of text that have been modified, the ratio and frequency of changes that happen to a web page, and so forth.

What to Notify –

We observed that many users prefer to have the WebCQ system track changes to the web pages of interest and notify them of the changes with some summarization in addition to individual change report per sentinel. In WebCQ, we provide support for the following four types of notification reports:

- Detailed sentinel evaluation report, displaying the differences side by side or in a merged document;
- A change summary report per user, displaying the list of sentinels installed by a user, each with a brief summary of the most recent change detection status. This approach is especially useful when the number of web pages being monitored per use is not small.
- A change summary report per user grouped by topic of the web pages being monitored. This approach scales much better than the previous one when the number of web pages monitored per user is getting large.
- A dedicated web notification site where a query interface is provided to allow users to search and find the change summary reports or the detailed change reports which match given conditions. This approach is particularly useful for busy users or users with Wireless devices and intermittent Internet connectivity.

The WebCQ notification service is utilized in at least three ways: First, we use the notification service to notify users of their change monitoring sentinel subscription, including

both the subscription of the WebCQ system, the expiration of their installed sentinels, and the expiration of their WebCQ subscription. Second, we use the notification service to send the users the update alert and the change summarization of the web pages being monitored. Third, the WebCQ notification service is designed to allow an external event observer to be loosely coupled with the change notification. Such coupling enhances both the tracking capabilities of the WebCQ system as well as the update monitoring functionality. It allows us to provide the WebCQ users not only the service for alerting and notifying users of the changes to Web pages of interest, but also the capability of taking appropriate actions to react to the changes. In the current implementation of WebCQ, we require the users to explicitly register the event observation function to be called and the method of remote invocation when a notification service is subscribed separately from the WebCQ system.

3.9 Optimization with Sentinel Grouping

In WebCQ, ideally there should be no restriction on how many Web page sentinels a user can create and how many users can register with the system. However, the system scalability and performance problems may arise when tens of thousands or millions of sentinels (continual queries) are running concurrently (500 users with an average of 20 sentinels per user will reach ten thousand CQs). Thus, one of the main challenges that WebCQ needs to address is the *scalability* of the sentinel processing strategy, namely how to guarantee the responsiveness of a Web monitoring service in the presence of a large number of concurrently running sentinels (continual queries).

Sentinel Grouping is an optimization technique developed in WebCQ. It promotes the reduction of repeated computation and processing by classifying sentinels into groups. Sentinels can be grouped together if they share the same target and have similar trigger conditions or their sentinel types and sentinel objects can be processed in one-pass. Consider a simple example. Assume we have n users who want to monitor new cancer trails over the National Cancer Institute (NCI) Web site. Each user is interested in monitoring the new cancer trails of m types of cancers separately. Thus we need to run $n * m$ Web page sentinels. Let T denote the time unit for the system-controlled polling interval (e.g., every 2

hours). Assume that k is the number of distinct types of cancers being monitored by the n users. If we group all $m * n$ installed page sentinels by the cancer type, then the number of concurrent trigger testing against the remote NCI Web site (www.nci.org) can be reduced from $m * n$ (one per sentinel) to k. Assuming a reasonable overlap of user interests in the types of cancers, such as breast cancer or prostate cancer, k is often significantly smaller than $m * n$, thus sentinel grouping will provide the WebCQ system a dramatic saving on both the network cost and the overall system overhead.

3.9.1 Grouping Strategy

The key idea behind the sentinel grouping is based on the general premise that a large number of Web page sentinels are often similar in terms of the target URLs, sentinel types, or sentinel objects. We distinguish three different classes of sentinels.

- The first class of sentinels are those that monitor the same Web page for any change or for a change on the same sentinel type and the same sentinel object. These sentinels are often captured in the same continual query (CQ) expression. We call this category of sentinels *the sentinels of the same CQ expression.*

- The second class of sentinels refers to those that monitor the same Web page (i.e., with the target URL) but monitor different fragments (i.e., different sentinel objects) in a page with the same sentinel type (e.g., table sentinel). These sentinels are captured in different CQ expressions. We call this category of sentinels *the sentinels of different monitoring objects.* For example, some students may be interested in monitoring the list of senior 4000-level courses in the course listing web site of College of Computing at Georgia Tech, whereas others may want to monitor 8000-level graduate courses in the same page.

- The third category of sentinels refers to those that may track changes in the same page but with different sentinel objects of interest and different sentinel types of interest. For example, some may want to monitor the CS4400 course home page for changes in the course schedule table, and others may be interested in monitoring the same page

for changes in the paragraph on grading policy or the list of reference textbooks.

In this section we discuss the index structure for sentinel grouping in WebCQ and the implementation method based on the concept of sentinel signatures.

3.9.1.1 Index Structure for Sentinel Groups

In WebCQ the index structure we use to store sentinels is a hash table with the URL of the monitored page as the hash key. Each bucket of the hash table contains a list of groups, one group per URL. Sentinels that share the same target URL are hashed into the same bucket accordingly. Figure 26 shows a sketch of the data structure used for grouping sentinels of similar CQ expressions. Each group has a corresponding URL, an MD5 hash of the most recently cached version of the web page, and pointers to the nine sentinel types specified in Section 3.3. Each pointer points to a linked list of n_i constant tables for the corresponding sentinel object type i. A constant table contains a set of sentinel identifiers that share the same URL target, the same sentinel expression, and the same table fragment being monitored. For example, the pointer to the list sentinel type ST_5 may have a linked list of n_5 constant tables and each refers to one concrete `List` fragment being monitored in the page.

Whenever a new page sentinel is installed, the system will first capture it in a continual query expression. Then this sentinel will be hashed into the corresponding bucket, depending on its target URL and its sentinel type. A constant table is created for this sentinel with an entry added to record the sentinel identifier, the corresponding CQ expression, among others, for this newly installed Web page sentinel. If the target URL is new, or the bucket for this sentinel type has not been created, it will first create the MD5 hash signature of the page and then create the bucket for the sentinel type.

Concretely, sentinel groups in WebCQ are constructed as follows:

Step 1: The URL for each sentinel is hashed into a bucket. If the sentinel is the first entry in the bucket, a new group is created, and the sentinel is added to the group, as described in step 3.

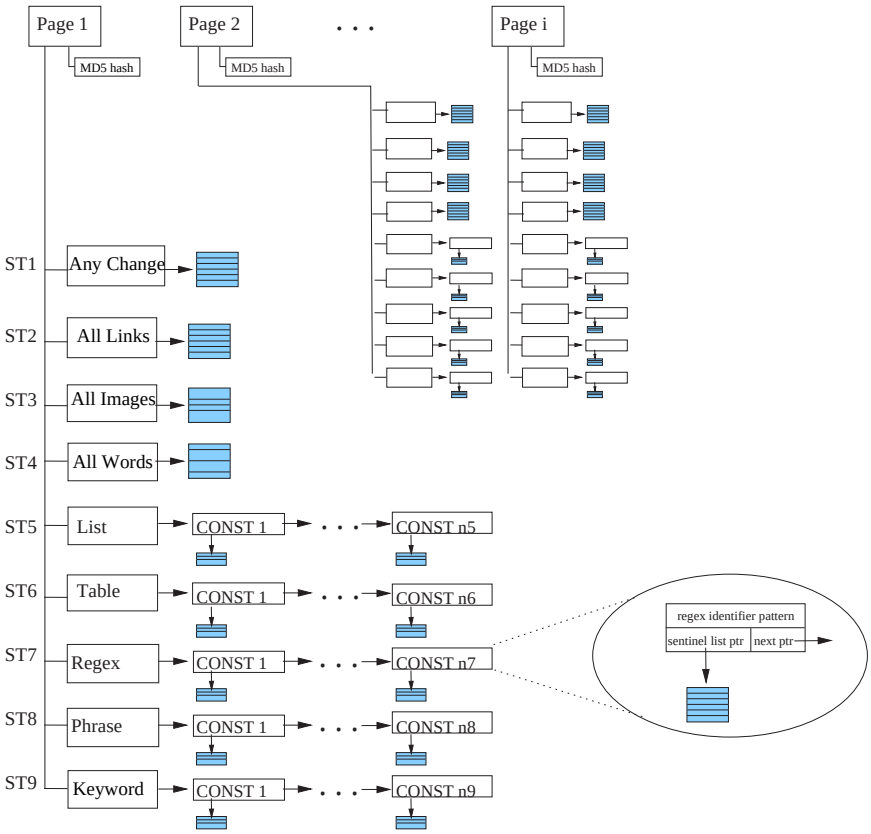

Figure 26: Web Page Sentinel Grouping Strategy

Step 2: The page is then retrieved from the proxy and an MD5 hash is computed for the page signature.

Step 3: If there are other entries in the bucket, the sentinel is added to a pre-existing group that was already setup to monitor that URL.

- If the sentinel is an `anychange` sentinel, the sentinel ID is added to the `anychange` sentinel table in the index structure.
- If the sentinel is a table, list, paragraph, or regular expression change sentinel, the specific sentinel specification must be compared with each constant in the linked list of constants corresponding to that sentinel type. If a match is found, then the sentinel ID is added to the corresponding constant table of sentinels. If no match is found, then a new constant identifier is generated, and inserted into the linked list of constants for that sentinel type. A constant table is created

accordingly, and the sentinel ID is added to this newly created constant table.

An obvious extension of the sentinel grouping strategy described in this section is to explore the alternatives for grouping sentinels. For example, This is especially desirable when a large number of CQs all have different trigger conditions but share some partial selection predicates. One idea is to choose the most selective predicate as the key predicate for grouping. We will discuss this topic in the next section.

3.9.1.2 Executing Sentinels Using the Group Index

Grouping sentinels over the same page can greatly reduce redundant operations. The process of executing grouped sentinels, managed by the WebCQ Daemon, operates in the following manner.

- **Step 1:** A URL is chosen from the list of all monitored URLs, and the current version of the page is requested from the proxy.
- **Step 2:** The sentinel group for the URL is retrieved from the metadata manager.
- **Step 3:** An MD5 hash is created for the page as the new page signature, as described in Section 3.9.2, and compared with the previous signature of the page. If the signatures match, no further processing is required for this group.
- **Step 4:** The previous version of the page is loaded from disk.
- **Step 5:** If there is any `anychange` sentinel (or all images, all links, all words sentinel) installed over this page, the old version and the current version of the page are compared to detect difference; if no change is found, then the processing for this group ends. Assuming a change is discovered, all `anychange` sentinels are notified.
- **Step 6:** Similarly, for the rest of sentinel types, check the linked constant list and for each constant table in the linked list, repeat the following process for each sentinel identifier:
  (1) load the object cache for the sentinel object; (2) locate the corresponding sentinel object in the current page, (3) compare the new version of the sentinel object with

the cached version for difference, and if a change is found, call the trigger evaluation module or Change summary generation module, otherwise no change is found, and exit.

An independent garbage collecting process is provided to clean up the local cache, and remove versions of pages that are older than the current version.

3.9.2 Generating Page Signatures Using MD5

Change detection can be considered as pure overhead when comparing identical versions of a web page. Even in character-based difference algorithms, such as Unix `diff`, each character of the page must be compared for each new version of the page. Many HTML or XML difference algorithms require the Web document to be converted into a tree structure before applying the diff algorithm. When nothing has changed, the time spent to convert a page from text into an in-memory model (such as a DOM tree), to load the previous copy of the page into memory from disk, and to compare the new page with the cached copy, is completely wasted.

To eliminate the unnecessary local I/O and diff comparison between two identical versions of a page, a signature for each page can be kept in memory with the sentinel group for that page. Then, when a new version of the page is retrieved from the web, a new signature is computed and compared with the signature of the previous version of the page. Only if the signature has changed does the local version need to be loaded into memory to determine what has changed to the page.

Typically high quality signature algorithms, such as MD5, are somewhat time-consuming to compute. In order for signatures to provide a net benefit, it should be at least cheaper to compute a signature for a page than it is to load the page from disk into memory. The probability that the page has changed should also be lower than the ratio of time to compute the signature to the loading time for the page. Signature computation can be done in parallel on multiple processors or computed while other processes are blocked on I/O. In our initial experiments, computing an MD5 signature is less than one-third as expensive as loading a DOM tree of the page from disk, providing a net benefit to using the MD5

signature algorithm on pages that do not change on each request.

3.10 Experimental Results

In this section we report four sets of experiments that evaluate the performance of the WebCQ change detection algorithms and sentinel grouping method. The first set of experiments compares the performance of different types of sentinels. The second set of experiments examines the costs associated with sentinel grouping. The third set of experiments demonstrates the benefits gained from sentinel grouping. The fourth set of experiments reports the performance comparison of the WebCQ original implementation with the WebCQ grouping implementation.

All experiments were run on a SunFire 280, dual 733 MHz processor server, with 8GB RAM, 72 GB disk on a RAID 5 controller, and gigabit local Ethernet. The software was implemented in Java and run with the J2SE 1.3.1 from Sun, using the server virtual machine. All experimental results discussed in this paper were averaged over more than ten execution runs over the chosen test data sets.

The set of data used for these experiments were pages gathered from the Yahoo! news portal and `my.yahoo.com` during December of 2001. These pages are chosen for our experimental evaluation as they represent relatively complex web pages, and they support a large variety of sentinel types, and are constantly changing. Page size in this set varies between 30KB and 60KB, averaging 47KB; the number of nodes in each page varies from 1400 nodes to 2171 nodes, averaging 1672 nodes.

3.10.1 Performance Comparison of Different Types of Sentinels

A major component of the WebCQ system is the algorithms used to detect changes in web pages by comparing the previous version of a page with the current version after both versions are loaded to memory. Figure 27 shows the execution time over the randomly generated data set of four different sentinel types: (1) any change sentinel over content using character-by-character comparison (any change sentinel for short), (2) link sentinel, (3) table sentinel, and (4) any change sentinel using MD5 signature (MD5-based any change sentinel). The any change sentinel monitors the entire content of a page. It parses the page

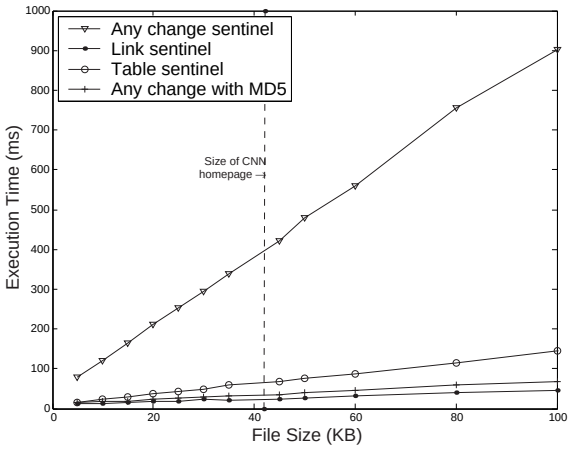

Figure 27: Processing time for detecting changes for different type of sentinels

character by character like the Unix `diff` utility and when a change is found, it returns the location of the object that has been changed. The link sentinel scans the page, locates all of the links, and checks to see if any have been added or removed from the page between versions. The table sentinel locates the table sentinel object in the web page using a regular expression, and compares this current version of the table sentinel object with the same table from the previous version of the page. If a change is found, it returns the location of the object that has been changed. The MD5-based *any change* sentinel computes hashes for the current version of the document, and compares it with the cached MD5 signature of the previous version of the page. Although technically there is a possibility for any hash function that two separate documents may have the same hash value (MD5 signature), it is well known that MD5 has extremely small probability for hash collision, especially for two versions of the same document. One benefit of the MD5-based *any change* sentinel over the original *any change* sentinel is that the signature of a page may be kept in memory, and the previous version of a page need not be loaded if the signature for the new version matches the signature of the previous version cached in memory. Figure 27 shows two interesting observations. First, some sentinels are inherently more costly than others, but the cost for all sentinel types increases uniformly with the size of the page being monitored. Second, the experimental results validate the intuitive result that the MD5-based *any change* sentinel performs significantly faster than the *any change* sentinel does.

3.10.2 Cost for Grouping Sentinels

While grouping sentinels together can save memory and execution time, there is some overhead for creating groups. Sentinel grouping is performed in three steps: creating a group, inserting new sentinels into a group, and loading and computing the MD5 hash for the group. Figure 28 and Figure 29 demonstrate the relative costs of grouping sentinels depending on the group size. The pages used for this experiment were based on the data collected from *Yahoo!*.

Figure 28 shows that there is a cost for grouping sentinels. When the group size is one, namely no sentinel is similar and thus can be grouped together, the cost of grouping increases linearly with respect to the number of sentinels. As the size of the group increases, the cost of grouping drops quickly. With group size of 10, the grouping cost is already considerably low. The cost of grouping sentinels decreases for larger group sizes because adding a sentinel to a group is the cheapest of the three steps, while the other two steps only need to be performed once per group. The cost of creating a group and computing the MD5 hash signature for the target page associated to the group can be amortized over the size of the group.

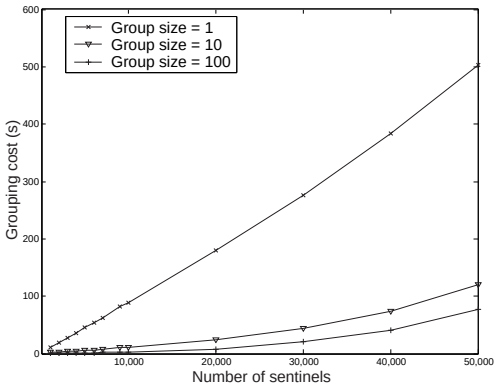

Figure 28: Grouping cost for varying group sizes

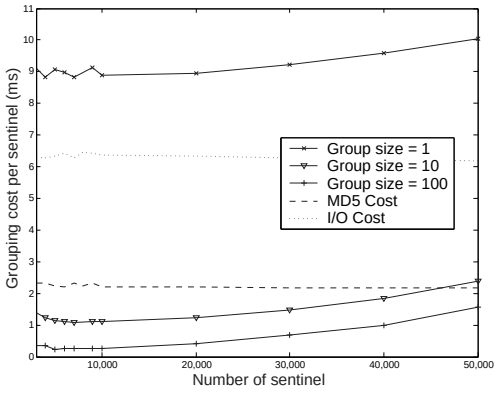

Figure 29: Grouping cost per sentinel

Figure 29 shows the grouping cost per sentinel. There are five lines in Figure 29: three lines listing the average cost of grouping sentinels into groups of different sizes, and two lines showing the average cost of the local I/O required to retrieve a document and the cost to compute an MD5 signature for a document. As expected, the average time required

to group sentinels hold steady as the total number of sentinels increase. Group time for a particular sentinel depends on the average number of sentinels per group rather than the total number of sentinels in the entire system. This behavior is desirable for any system that must scale to millions of sentinels.

3.10.3 Performance Gains for Sentinel Grouping

The third set of experiments measure the benefits of sentinel grouping. These experiments were run over pages from the data set gathered from *Yahoo!*. Figure 30 shows the cost of executing any change sentinels without any form of grouping. It is clear from this experiment that the I/O time required to load the two versions of a Web document to be compared is very high compared to the overall execution time.

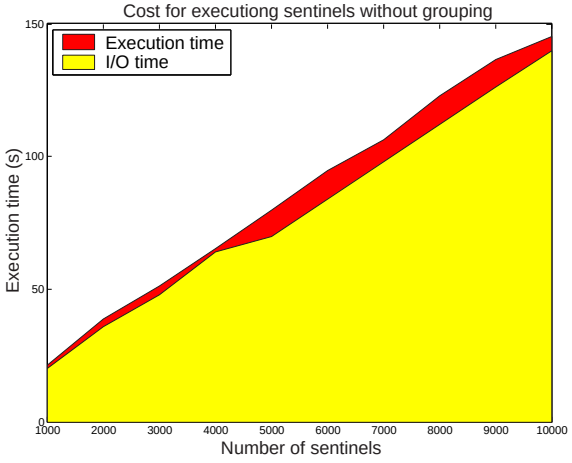

Figure 30: Execution time without grouping

While I/O is obviously the major factor in sentinel evaluation, in the next experiment we want to show that sentinel grouping may significantly reduce the costs in evaluating sentinels for grouping with both uniform distribution and Zipf-distribution. It is expected that Web-page sentinels be distributed in a Zipf-like pattern, in which a few pages have many sentinels whereas most pages have only a few sentinels installed over them. Zipf-like distributions are expressed by the equation Ω/i^{α}, describing the popularity of page i, where Ω is a normalizing constant, i is the i^{th} most popular page, and the exponent α describes the slope of the curve. This is typically the distribution seen by general proxy servers [17].

Figure 31 compares grouping effects for Zipf-like distributions of sentinels on web pages with no grouping and grouping where the group size (gs) is ten.

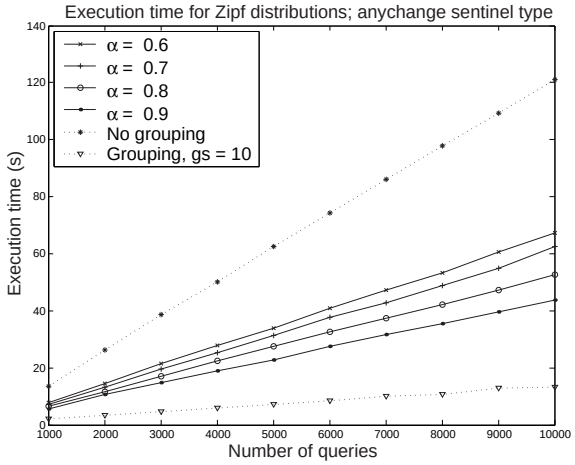

Figure 31: Grouping with Zipf and uniform distributions

Figure 32 shows the total time required to execute several thousand triggers with no grouping or grouping with 5, 10, and 100 sentinels per group under uniform distribution. For 5,000 sentinels, the total execution time with a group size of 5 is 14.2 seconds, while it is 7 seconds for a group size of 10 and 1.3 seconds for a group size of 100 sentinels. Without grouping, however, it takes over 62 seconds to process the same number of sentinels.

Figure 33 shows the throughput for various levels of grouping. For 5,000 sentinels, without grouping WebCQ can process 80 sentinels per second. With grouping, the WebCQ system achieves throughput rate of 351 sentinels per second when group size is 5, over 4 times more sentinels can be processed per second. Similarly, the throughput rates are 708 sentinels per second for group size of 10, and 3,698 sentinels per second for group size of 100, which are almost 9 times improvement for group size 10 and over 46 times improvement for group size of 100.

3.10.4 Comparing WebCQ with Grouping to WebCQ Original

Work on sentinel grouping in WebCQ began due to several limiting factors in the first implementation of WebCQ. The original WebCQ was a novel implementation of a change detection system for web pages. It incorporated several ideas from our earlier work on the

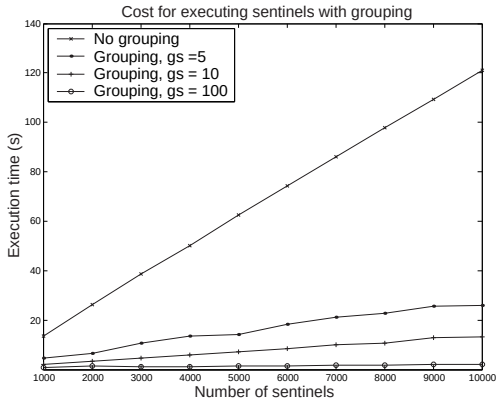

Figure 32: Execution time for grouped sentinels

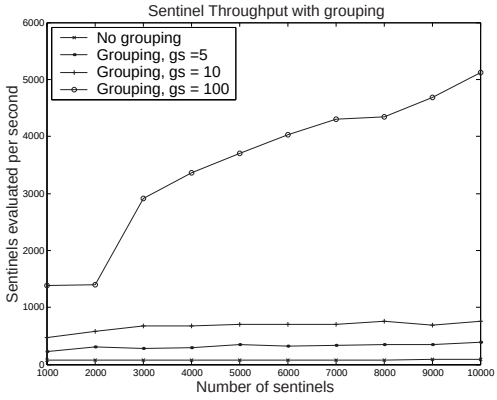

Figure 33: Throughput for grouped sentinels

CQ project [64], applying the concept of continual queries to data available over the web.

However, the first implementation of WebCQ consumed too many resources in managing and executing sentinels. The system relied on the Oracle 9i database system to manage all of the metadata, including user profiles, sentinels, and the object cache. Each sentinel execution required multiple accesses to the database. Execution of individual sentinels relied on external tools that were not optimized for large scale processing. To alleviate the problem, in WebCQ grouping implementation we moved the management of most information out of the database and into main memory. The historical data cache was moved to the file system where we also keep extensive logs to guard against system failures. Furthermore, we implemented our own change detection and management tools directly in the system architecture, providing a seamless intraprocess workflow for all operations. The combined improvements in sentinel algorithms, I/O sensitive operations, and sentinel grouping have contributed to achieving one to two orders of magnitude improvement over our initial version, as documented in Figure 34.

3.11 Discussion

We have presented the design and development of the WebCQ system, including the sentinel grouping technique for scalable processing of Web page sentinels. We also reported our initial experimental results showing the benefits and the effectiveness of the WebCQ approach to large-scale information monitoring on the Web. In this section we discuss a

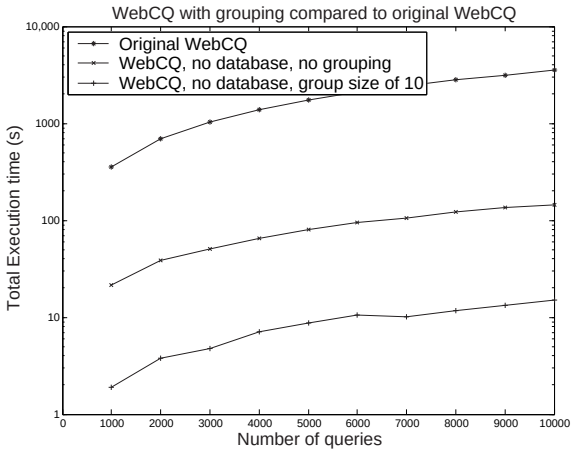

Figure 34: WebCQ with grouping vs. original WebCQ

number of important ongoing issues and our plan for extensions along several dimensions.

3.11.1 Monitoring and Notification Latency

Latency is an important factor in WebCQ information monitoring and delivery service. In WebCQ we use the concepts of change detection interval and notification interval to model the overall monitoring latency. Typically, a *source-specific change detector* (whether it is source-initiated or polling-based) sets the sampling latency, bounding the delay from source event occurrence until change events are detected. We refer to such delay as the *change detection interval* or the *sampling latency*. This "change detection interval" determines the minimum resolution of event frequency. The notification manager controls the latency from change detection until notification. We refer to such latency as the *notification latency* or *notification interval*, which determines the maximum resolution of notification frequency.

Achieving low sampling latency (less than notification latency) may require source-initiated change detection. Our experience with tracking changes in Web pages shows that it is hard to decide what polling intervals are the most appropriate or efficient. Inappropriate setting of polling interval is one reason polling-based change detection systems can be inefficient. But when source-initiation is unavailable, which is the case for most web sources, the polling approach is the only choice.

There are several factors affecting the notification latency of the WebCQ system. First,

in determining whether the change event source or the notification server should initiate notification delivery, one needs to know which party has enough information on its counterparts. If an event source knows its listeners, the source can initiate notification delivery upon event observation. For example, a printer can send a message to the owner of the print job upon print completion (in Unix, this is achieved by simply using "-m" option for *lpr*). However, if an unknown set of users want to monitor a single source (e.g., the current job queue of some printer), a client-polling policy is more appropriate. For example, Unix users can use "lpq" to list a printer job queue.

Second, change notifications can be sent directly from a source to the end-receivers, or through an intermediate server using either "end-to-end" delivery or "store-and-forward". In very large-scale environments such as the Internet, the use of intermediate proxies is common.

Third, in terms of notification delivery constraints, the *Notification Service* is responsible for ensuring certain guarantees before delivering a notification to the recipients. For example, whether the notification supports real time delivery or not, or whether to resend when messages are lost. Currently, we use a "no retry" policy in the WebCQ system for simplicity. Performance improvements on reducing both the sampling and the notification latency are under way. In addition, security constraints are also an important responsibility of the notification service. For example, in case of a firewall, the service needs to have appropriate knowledge of how to pass messages through the firewall, for example via proxies.

Finally, to ensure correct notification, the notification manager maintains a subscription list, which is an editable and auditable "first-class" object. An initial verification is performed at the time of a notification service subscription. For example, in WebCQ, the email addresses are checked upon user registration in order to guarantee that the correct email addresses are used for the users to receive notification. We are interested in incorporating more advanced quality of service (QoS) properties into the notification service of the WebCQ system.

3.11.2 Scalability

According to B.C. Neuman [79], "the scale of a system has three dimensions: numerical, geographical, and administrative. The numerical dimension consists of the number of users of the system and the number of objects and services encompassed; the geographical dimension consists of the distance over which the system is scattered; the administrative dimension consists of the number of organizations that exert control over pieces of the system."

It is widely recognized that the scale can affect many components of the system: naming, authentication, authorization, accounting, communication and the use of remote resources. Scale also affects reliability: when a system scales numerically, the likelihood that some host will be down increases; when a system scales geographically, the likelihood that some hosts fail to communicate with others increases. Scale can affect performance as well, in terms of system load and communication latency.

General solutions to scalability fall into four categories: replication, distribution, caching [79], and optimization. They can be both hardware-related and software-related. We considered scalability when we began to build the WebCQ system:

Replication : WebCQ can replicate the meta database (user info + registered WebCQ) to improve the response time and availability of the whole system (under development). The placement of replicas is an ongoing research project.

Distribution : WebCQ can direct different user requests to different servers. One example is to distribute using a hashing function on user's email addresses. Another example is to distribute according to different user request domain, e.g., stock watch, book watch, and flight price watch. Some software components can also be spread across multiple servers, such as email services (under development).

Caching : WebCQ system uses cache proxy architecture to increase scalability. When the number of users and number of user requests increase, it is possible that different requests have interests in the same Web page. Web caching has been used for quite some time in general Web server and browser architectures. In WebCQ, we cache the accessed Web pages and store them locally for later use. We use a policy file to control

the frequency of refreshing the pages. The default is one day. When we serve a new user request, we first look locally in the cache for the page, if it is within the refresh threshold, we retrieve it from the cache, otherwise, the proxy server fetches the new content from the source site. From our experience in building the Continual Query system, we assume local access is always faster than communicating with remote servers.

Optimization : WebCQ develops the sentinel grouping techniques that optimize the concurrent processing of large number of sentinels by grouping sentinels of similar structure together to reduce the duplicated computation. Our initial experimental results (recall Section 3.10) demonstrate the effectiveness of sentinel grouping for scalable processing of sentinels.

3.11.3 Availability

The WebCQ system is a Web-based middleware system. It involves Web servers, proxy servers, and other software components. The whole system can be viewed as a cluster of software components. The technical challenge and the attempts made in WebCQ for ensuring availability can be summarized as follows:

- No bottlenecks to scaling: new servers can be added and configured dynamically; requests can be distributed to multiple software components.
- No single points of failure that could impact availability: requests must automatically failover to working components (by replication).
- Transparency to applications and application developers: the system software takes care of the replica management and consistency control.
- Single-system image to system administrators: the cluster is managed as a single logical resource.
- Hardware and operating system independence: WebCQ uses Java to achieve maximum software portability and independence

The WebCQ system has a presentation front end (user interface) and a back end (server end). For the front end, different mechanisms are used to increase the scalability and availability. For example, one commonly used approach to direct user request to different servers is "DNS Round Robin" between the Web clients and the Web servers. This provides simple form of load balancing and fail-over schemes. Another clustering technique is for dynamically generated pages that go between the Web server and the Java Servlet engine. A Java servlet is capable of keeping a user session or a persistent database connection (to avoid database connection setup cost). Application states can be stored in the server meta database (which can be replicated). However, there are many open issues in this endeavor. One of our ongoing research efforts is to investigate the state of art in availability research and develop a general solution for improving the availability of information monitoring systems such as WebCQ.

3.12 System Status

The *WebCQ* project was started in 1999 when I moved with the CQ group to Georgia Tech. The first prototype was done in early 2000 and made public in May 2000. The system was a total rewriting of the previous *OpenCQ* to handle change monitoring requests for arbitrary Web pages. The system is currently hosted at the College of Computing in Georgia Tech. It can be accessed online at http://disl.cc.gatech.edu/WebCQ. A set of toolkits are also available from the site. The system is published as open source software in November 2003.

CHAPTER IV

DIFFERENTIAL EVALUATION ALGORITHM FOR CONTINUAL QUERIES

In Chapter 2 and Chapter 3, I have described the general optimization technique for evaluating *Continual Queries*, namely the *Continual Query (Sentinel) Grouping* technique. In this chapter, I am going to introduce another technique for optimizing continual query executions. The work presented in this chapter is based on previous research work [60, 66].

4.1 Motivation

Continual queries are standing queries that run continuously until the termination condition becomes true. Whenever an relevant update is performed, the CQ system will trigger the execution of the corresponding continual queries. It is obvious that the subsequent execution of a given continual query is only interested in those data that have been updated since the previous execution. In the situation where the amount of updates is small, one way to optimize the subsequent executions of a given CQ query is to use differential evaluation method such that queries that can be answered using delta information (i.e., the updated data) rather than the full set of base data. Similar techniques have been widely used in incremental view materialization [14, 44, 59, 52, 88].

More concretely, recall Chapter 2, we have defined a CQ query as a sequence of query results, modelled by $Q_{cq}(S_1)$, ..., $Q_{cq}(S_n)$, where $Q_{cq}(S_i), (i = 1, ..., n)$ is the result of running Q_{cq} on the database state S_i. Quite often there are situations where users are more interested in the difference between $Q_{cq}(S_i)$ and $Q_{cq}(S_{i+1})$. This can be accomplished by naively executing the entire query and then filtering out the part of the query result that is the same as the previous result. This simple and straightforward approach can be quite expensive, especially in the Internet environment where query results need to be gathered from multiple source data repositories. An obviously more attractive approach

is the differential query evaluation method, which is particularly powerful when $Q_{cq}(S_i)$ is relatively large, and only a small percentage of the result changes from state S_i to state S_{i+1}.

For example, suppose we have a join query $R \bowtie S$ and let us assume for simplicity that R is located at site 1 and S is located at another site, namely site 2. Suppose this query is installed as a continual query that runs every 5 hours. It means that the CQ server will check out the updates at the source R and S every 5 hours. Whenever R or S objects at the sources change, the query $R \bowtie S$ will be reevaluated again. Suppose we have a new object o_i added into the source R at time t_1 after the initial installation of this CQ at t_0, $t_1 - t_0 < 5h$, and this insertion is the only update at source R and source S up to the time point $t_0 + 5h$. Obviously the change effect can be computed by either executing $R \bowtie S$ again and then computing the difference with the previous result, or by computing the net change effect using $\{o_i\} \bowtie S$. When the cardinality of R is very large, the differential computation of net change effect is much cheaper than the re-evaluation of the original query expression $R \bowtie S$ from scratch.

[60] proposes a differential re-evaluation algorithm for continual queries. The key idea behind this algorithm is to produce $Q(S_{i+1})$ by incrementally updating $Q(S_i)$. More concretely, in contrast to a complete re-evaluation, differential re-evaluation means that after the initial execution of a CQ, the re-evaluation of each subsequent execution of this CQ will be performed by using the differential form of the query, denoted as δQ_{cq}. This way, we avoid reprocessing the entire query from scratch. When the changes are substantially smaller compared with the latest query execution result, this differential update will be more efficient than reprocessing the entire query.

The differential re-evaluation algorithm (DRA) is invoked by the CQ manager based on the epsilon specification associated with the given CQ. We assume that the information available when the DRA is invoked includes:

- the CQ specification (Q_{cq}, $Trig_{cq}$, $Term_{cq}$);
- the contents of each base relation after the last execution of the CQ;

- the differential relations for each of those operand relations that have been changed since the last execution of the CQ;
- the timestamp of the last execution of the CQ;
- the complete set of the result of the CQ produced by the last execution.

In short, the Differential Re-evaluation Algorithm (DRA) is developed for incrementally computing the new query result from processing updates on top of the previous result. [66] proves that the differential re-evaluation algorithm - DRA is *functionally equivalent* to the "recompute the query from scratch" solution, and, in many situations is more efficient.

Although the differential re-evaluation of continual queries has been extensively covered in [60, 66], to make the discussion of our implementation design of the DRA easier to understand, and make this thesis self-contained, in this chapter we will first present the notation and an brief overview of the differential reevaluation algorithm, and then describe the design choices and implementation plan that we have laid out for efficient realization of the DRA in the first prototype of the CQ system.

The structure of this chapter is as follows: Section 4.2 describes the notation and terminology required to explain the DRA algorithm. Section 4.3 describes the strategy to generate $Q(S_n)$ from $Q(S_{n-1})$ incrementally, thus reducing both processing time and network transmission bandwidth. Section 4.7 discusses the implementation consideration of the DRA algorithm.

4.2 Notations and Terminology

The relational terminology is used in this paper to specify continual queries, to record and manipulate changes by differential relations and associated operators. This notation does not constrain our algorithm and solutions to relational database management systems. In DIOM, information consumers formulate queries with GUI tools, which are then translated into appropriate query languages for backend processing, such as SQL. On the other side, information producers need only to generate the differential relations, which are simple tables of update operations, to communicate with the consumers.

We assume that the reader is familiar with the basic concepts and notation concerning relational database, as described in [72]. In this paper we refer to relational selection, projection, join, outerjoin, union, and difference operators by σ, Π, $\bowtie$, `Outerjoin`[1], $\bigcup$, and $-$ respectively. For presentation convenience, we sometimes use $\Pi(R;X)$ to denote $\Pi_X(R)$, $\sigma(R;F)$ to denote $\sigma_F(R)$, and a SPJ expression to denote a `Select-Project-Join` expression.

We use differential relations to represent the net effect of a collection of updates to a relation, either stored or derived. The differential relation for a stored relation is instantiated by the system when the source is updated by insertion, deletion or modification (see Example 4). We define an operator that computes differential relations for arbitrary SPJ expressions. The concept of differential relations is, to some extent, similar to the concept of hypothetical relations used for incremental updating materialized views[14, 44]. The difference lies in the usage and the detailed structure. In the eager mechanism for materialized view update, a hypothetical relation represents the net changes made by a single transaction to a base relation and can be dropped after the transaction is committed and the materialized view is updated. In the continual query refresh method, a differential relation actually maintains changes made by several transactions to a base relation. Data in the differential relation can only be dropped when their timestamps are "older" than the timestamp of the latest execution of *every* relevant continual query.

We would like to mention that the relational model is not essential to our approach, but it simplifies the representation of database changes, allows use of the relational algebra, and avoids the need to explain the semantics of a particular object model.

4.2.1 Differential Relations

We introduce the concept of differential relation, a relation that can represent changes to another relation, *and* design a set of basic operations to facilitate the manipulation of such relations. The goal for defining differential relations, instead of using hypothetical relations

[1] `Outerjoin` keeps all tuples in both the left and right relations when no matching tuples are found, padding them with `null` values as needed [72].

described in [14, 44], is to provide a unified treatment of changes, not separate treatments of insertions, deletions, and modifications resulting in several algorithms for generating and combining individual results.

Let $\mathcal{R}$ denote a relation scheme described by a set of attributes $A_1, A_2, \ldots, A_n$. Let R denote a relation instance of $\mathcal{R}$ consisting of a collection of tuples whose values are taken from the domains of the set of attributes of $\mathcal{R}$. $t.A_i$ $(1 \leq i \leq n)$ denotes the value of attribute A_i for tuple t. Each tuple has an attribute, denoted as *tid*, which provides a unique immutable identifier. When a tuple is deleted and later re-appended to R, it will have a new *tid* assigned. The unique tuple identifier *tid* makes it easier to connect tuples that hold values of the same object before and after changes. In fact, the primary key can be used as the unique identifier for each tuple of R.

Definition 1 (Differential relation)
Let $\mathcal{R} = (tid, A_1, A_2, \ldots, A_n)$ be a relation scheme. For each relation R of scheme $\mathcal{R}$, we define a differential relation, denoted by ΔR, to represent changes.

$$\Delta R = (tid^{<old>}, A_1^{<old>}, \ldots, A_n^{<old>}, tid^{<new>}, A_1^{<new>}, \ldots, A_n^{<new>}, \texttt{timestamp}),$$

where

- $A_i^{<old>}$ refers to old attribute values and
- $A_i^{<new>}$ $(1 \leq i \leq n)$ refers to new attribute values.
- For any tuple $t \in \Delta R$,
  - $tid^{<old>}$ and $tid^{<new>}$ cannot both be null.
  - If $tid^{<old>}$ is null, so are all $t.A_i^{<old>}$ $(1 \leq i \leq n)$.
  - Similarly, if $tid^{<new>}$ is null, so are all $t.A_i^{<new>}$ $(1 \leq i \leq n)$.
  - If both $tid^{<new>}$ and $tid^{<old>}$ are not null, then $tid^{<new>} = tid^{<old>}$.
- For each tuple in ΔR, the system assigns a `timestamp` at its creation time as its identifier.

Differential relations can represent tuples (or objects) where only the tid field is non-null. No tid can appear in multiple rows. For addition, the attributes $A_i^{<old>}$ $(1 \leq i \leq n)$ are null. For deletion, the attributes $A_i^{<new>}$ are null. For modification, attributes $A_i^{<old>}$ hold old values and attributes $A_i^{<new>}$ hold newly modified values. The `timestamp` field is set to the current time (from a system clock, or any other monotonically increasing source of timestamps) whenever a tuple is appended to ΔR.

Example 4 Consider the relation `Stocks` with attributes such as name and price per 100 units.

Stocks:

tid	Name	Price
120992	DEC	150
092394	OLI	145
032090	ODI	120
041977	USL	100
⋮	⋮	⋮

Assume that the transaction `T` updates the `Stocks` relation by insertion, deletion and modification:

```
    Transaction T {
    Insert (101088, MAC, 117);
    Modify (120992, DEC, 150) = (120992, DEC, 149);
    Delete (092394);
}
```

The following differential relation Δ`Stocks` captures the changes that the transaction `T` made to `Stocks`:

ΔStocks:

tid$^{<old>}$	Name$^{<old>}$	Price$^{<old>}$	tid$^{<new>}$	Name$^{<new>}$	Price$^{<new>}$	timestamp
-	-	-	101088	MAC	117	10
120992	DEC	150	120992	DEC	149	25
092394	OLI	145	-	-	-	50

In the rest of this section, for presentation clarity the field `timestamp` is omitted when no confusion is incurred. We sometimes simply use "delta relations" to refer to differential relations.

4.2.2 Basic Operations

We first define renaming functions that add or remove the superscripts *old* and *new* from attribute names in a relation scheme $\mathcal{R}$.

Definition 2 Let $\mathcal{R} = (tid, A_1, A_2, \ldots, A_n)$ be a relation scheme and R be a relation of scheme $\mathcal{R}$. Let the relation S be of relation scheme $\mathcal{S} = (tid^{<old>}, A_1^{<old>}, \ldots, A_n^{<old>})$. The differential relation schema of $\mathcal{R}$, is denoted by $\Delta\mathcal{R}$.

1. $\texttt{old}(\mathcal{R}) = \{tid^{<old>}, A_1^{<old>}, \ldots, A_n^{<old>}\}$.
2. $\texttt{new}(\mathcal{R}) = \{tid^{<new>}, A_1^{<new>}, \ldots, A_n^{<new>}\}$.
3. $\texttt{detach}[S; \texttt{old}(\mathcal{S})]$ returns the same relation with each attribute name detached from the superscript <old>.
4. $\texttt{attach}[R; \texttt{old}(\mathcal{R})]$ returns the same relation with each attribute name attached by the superscript <old>.

The second group of operators is used to project the *old* or *new* values of a differential relation, and to compute the updated relation, say R', by combining the relation R with its differential relation ΔR.

Definition 3 (High-level projection operators)

Let R be a relation of $\mathcal{R} = (tid, A_1, A_2, \ldots, A_n)$ and ΔR be a relation of the differential relation schema $\Delta\mathcal{R}$. We define the operators `deletions`, `insertions` and `combine` as follows:

1. $\texttt{deleteLog}(\Delta R) = \Pi(\Delta R, \texttt{old}(\mathcal{R}))$.
2. $\texttt{insertLog}(\Delta R) = \Pi(\Delta R, \texttt{new}(\mathcal{R}))$.
3. $\texttt{deletions}(\Delta R) = \texttt{detach}[\texttt{deleteLog}(\Delta R); \texttt{old}(\mathcal{R})]$.
4. $\texttt{insertions}(\Delta R) = \texttt{detach}[\texttt{insertLog}(\Delta R); \texttt{new}(\mathcal{R})]$.
5. $\texttt{combine}(R, \Delta R) = (R - \texttt{deletions}(\Delta R)) \bigcup \texttt{insertions}(\Delta R)$.

These high-level projection operators are derivations of the basic operations for differential relations. We define them here mainly for presentation convenience, because they are used frequently in our differential re-evaluation strategies for continual query processing.

Example 5 Consider the relation `Stocks` and the differential relation ΔStocks in Example 4.

deleteLog(ΔStocks)

tid$^{<old>}$	Name$^{<old>}$	Price$^{<old>}$
120992	DEC	150
092394	OLI	145

insertions(ΔStocks)

tid	Name	Price
101088	MAC	117
120992	DEC	149

Stocks' = combine(Stocks, ΔStocks)

tid	Name	Price
120992	DEC	149
032090	ODI	120
041977	USL	100
101088	MAC	117
⋮	⋮	⋮

4.3 Differential Evaluation of Continual Queries

In this section we present a differential re-evaluation algorithm (DRA) for processing a continual query efficiently. In contrast to a complete re-evaluation, differential re-evaluation means that after the initial execution of a CQ, the re-evaluation of each subsequent execution of this CQ will be performed by using the differential form of the query. The DRA is invoked by the CQ manager based on the epsilon specification associated with the given CQ. We assume that the information available when the DRA is invoked includes: (i)the CQ definition; (ii)the contents of each base relation after the last execution of the CQ; (iii)the differential relations for each of those operand relations that have been changed since the last execution of the CQ; (iv)the timestamp of the last execution of the CQ; (v)the complete set of the result of the CQ produced by the last execution. Note that the CQ manager will use (iv) to append the proper timestamp predicate(s) into the differential form of the CQ, which limits the search space to only those tuples that were written to the differential relations by the updates occurred after the last execution of this CQ (recall Example 6 for an illustration). We also formally study the correctness of the DRA with respect to the complete re-evaluation solution.

In what follows, we first formally define an operator that computes the differential relations for the data derived by arbitrary query expressions in Section 4.3.1. In Section 4.3.2 we introduce the differential forms for the three basic relational algebraic operations: `Select`, `Project`, and `Join`. We prove that instantiation of `Propagate`(Q; PL) for relational select, project, and join are functionally equivalent to their differential forms: `DiffSelect, DiffProj` and `DiffJoin`. The differential re-evaluation algorithm is described

in Section 4.3.3.

4.3.1 Computing the Differential Results for Continual Queries

In this section we introduce a high level operator, called `Propagate`, to describe how the result relation of a continual query changes when at least one of its operand relations changes. This operator computes the difference between two consecutive executions of a CQ by complete re-evaluation of the query for each execution. It can be considered as an instantiation of *complete re-evaluation* solution. The main purpose of introducing the operator `Propagate` is to formally prove that our differential re-evaluation algorithm to continual queries is *functionally equivalent* to the "recompute the query from scratch" solution, and, in many situations is more efficient.

Before giving the definition of the operator `propagate`, we define an operator that compute the difference of two relations of the same scheme.

Definition 4 (The operator `Diff`)
Let R_1 and R_2 be two relations of the same scheme $\mathcal{R}$. The operator $\texttt{Diff}(R_1, R_2)$ is defined as the $\texttt{Outerjoin}_{\texttt{tid}}$ of tuples in $R_1 - R_2$ and $R_2 - R_1$ under the join condition "$\texttt{tid}^{<old>} = \texttt{tid}^{<new>}$".

$$\texttt{Diff}(R_1, R_2) = \texttt{Outerjoin}_{\texttt{tid}}(\texttt{attach}[(R_1 - R_2);\texttt{old}(\mathcal{R})], \texttt{attach}[(R_2 - R_1);\texttt{new}(\mathcal{R})]).$$

It returns a differential relation ΔR that describes their differences. Null values are used to pad tuples that appear in only $R_1 - R_2$ or $R_2 - R_1$. The relation scheme of ΔR is $\texttt{old}(\mathcal{R}) \cup \texttt{new}(\mathcal{R})$.

Definition 5 (The `Propagate` operator)
Let R_i be a relation of scheme $\mathcal{R}_i$ $(1 \leq i \leq n)$, ΔR_i be a differential relation of R_i, and R_i' denote $\texttt{combine}(R_i, \Delta R_i)$. Let $\mathrm{Q}(R_1, \ldots, R_n)$ denote an arbitrary continual query and SL $= \{[R_i, \Delta R_i] \mid (1 \leq i \leq k,\, k \leq n)\}$ denote the differential substitution list. The `Propagate` operator is defined as follows:

$$\texttt{Propagate}(\mathrm{Q}(R_1, \ldots, R_n);\, \mathrm{SL}) = \texttt{Diff}(\mathrm{Q}(R_1, \ldots, R_n),\, \mathrm{Q}(R_1', \ldots, R_n')) \qquad (n \geq 1).$$

It returns the differential relation of query Q, denoted by ΔR_Q, with the scheme $\mathcal{R}_Q$.

We may also denote Propagate(Q($R_1, \ldots, R_n$); SL) by ΔQ($R_1, \ldots, R_n$).

Example 6 Consider the query Q: $\sigma_{\texttt{price}>120}(\texttt{Stocks})$ as a continual query. Let E_i be the ith execution of Q at time t_i and E_i(Q, t_i) denote the result of the ith execution of Q ($i = 0, \ldots, \infty$). Now assume that the base relation Stocks is changed, after the last execution E_i and before the current execution E_{i+1}, by the update transaction T given in Example 4. Let Q(Stocks) = E_i(Q, t_i) denote the result of the ith execution of Q over Stocks. Let Stocks' denote the base relation after updates to Stocks by transaction T, and Q(Stocks') = E_{i+1}(Q, t_{i+1}) denote the result of the current execution of Q, over the relation Stocks'.

Based on Definition 5, to express how the result E_i(Q, t_i) may change after the updates by the transaction T, we may simply compute Propagate(Q(Stocks);[Stocks,ΔStocks]), the difference between the result relation before the updates: Q(Stocks) and the result relation after the update: Q(Stocks').

Q(Stocks) = $\sigma_{\texttt{price}>120}$(Stocks) = {(120992,DEC,150), (092394,OLI,145)}.

Q(Stocks') = $\sigma_{\texttt{price}>120}$(Stocks') = {(120992,DEC,149)}.

Q(Stocks)−Q(Stocks') = {(120992,DEC,150), (092394,OLI,145)}.

Q(Stocks')−Q(Stocks) = {(120992,DEC,149)}.

The differential result ΔQ(Stocks) below represents the net effect made by all the updates occurred between the last execution E_i and the current execution E_{i+1}.

ΔQ(Stocks) = Diff(Q(Stocks),Q(Stocks'))

tid$^{<old>}$	Name$^{<old>}$	Price$^{<old>}$	tid$^{<new>}$	Name$^{<new>}$	Price$^{<new>}$
120992	DEC	150	120992	DEC	149
092394	OLI	145	-	-	-

In this example, the differential result of the query $\sigma_{\texttt{price}>120}$(Stocks), since the last execution of Q, is computed by directly evaluating the function: Propagate ($\sigma_{\texttt{price}>120}$ (Stocks); [Stocks,ΔStocks]). Assume the previous execution result E_i(Q, t_i) is saved. The evaluation of Propagate directly from its definition requires first a scan of the modified relation

Stocks' in order to compute Q(Stocks'). Such recomputing from scratch is often wasteful, and in many cases unacceptable.

Observe that when (i)the size of the source relation Stocks is large, (ii)the selectivity factor of the query Q over Stocks is not high, and (iii)the number of tuples written by the updates, occurred between the two consecutive executions E_i and E_{i+1}, is relatively small, it would be more efficient to compute the differential result of the query Q before and after the updates by evaluating the query over the differential relation ΔStocks instead. It is because computing Propagate($\sigma_{\text{price}>120}$(Stocks);[Stocks,ΔStocks]) is functionally equivalent to the evaluation of the differential query: σ_F(ΔStocks), where F denotes the predicate price$^{<old>}$ > 120 ∧ price$^{<new>}$ > 120 ∧ timestamp> t_i[2]. This is true even when the user needs the complete composite set of the query result, because computing the union of Q(Stocks) and insertions(σ_F(ΔStocks)) is cheaper than recomputing the expression $\sigma_{price>120}$(Stocks') from scratch.

Furthermore, using a differential evaluation approach, we can show those tuples that were removed between the two consecutive executions of Q, simply by computing deletions(σ_F(Δ Stocks)). In general, for any continual query Q over the relation R, let E_i(Q,t_i) be the last execution of Q at time t_i. A complete set of the result of current execution of Q can be obtained by computing the expression:

$$(E_i(Q,t_i) - \sigma_{timestamp>t_i}(\texttt{deletions}(\Delta R))) \cup \sigma_{timestamp>t_i}(\texttt{insertions}(\Delta R)).$$

The expression $\sigma_{timestamp>t_i}$(insertions(ΔR) returns all the records that have been appended to R since the last execution of Q; whereas the expression $\sigma_{timestamp>t_i}$ (deletions (ΔR) returns all the records removed from R since the last execution of Q.

4.3.2 Optimization based on Differential Operators

For many forms of queries, Propagate(Q; SL) can be computed more efficiently than by directly evaluating the definition of the Propagate operator. These computations use the differential relations heavily and sometimes exclusively, instead of computing on the base relations, which tend to be much larger.

[2]The condition timestamp> t_i is appended by the CQ manager for incorporating correct amount of updates and limiting the query search space. Section 4.6.3 provides some further discussion.

In this section we define the differential forms for the three basic relational algebraic operations: `Select`, `Project`, and `Join`. We prove that instantiation of `Propagate`(Q; SL) for relational select, project, and join are functionally equivalent to their differential forms: `DiffSelect, DiffProj` and `DiffJoin`. The predicates contained in these operators can be atomic or composite by logical "`AND`" and logical "`OR`". The attributes involved in the predicates or the projection list are single valued attributes. Aggregate functions such as `SUM, COUNT, MAX, MIN` are allowed in this framework with some additional consideration. For example, when the query expression contains the aggregate function `MAX`(A), we need to compare the A field of each $< new >$ tuple in the differential relation with the value of `MAX(A)`, and to reset `MAX(A)` if necessary. Similar treatment applies to `SUM, COUNT, MIN`. However when the query contains the aggregate function `AVG`, the recomputation of the query is needed.

4.3.2.1 The Differential Select Operator

Definition 6 (Differential select)
Let F denote a predicate defined on the relation R of scheme $\mathcal{R}$, and $\sigma[R;F]$ denote the associated relational selection operator. Let $F^{<old>}$ and $F^{<new>}$ denote the same predicate with all attribute names superscripted accordingly. We define the operator `DiffSelect` as follows:

`DiffSelect`$[\Delta R;F]$=`Outerjoin`$_{tid}(\sigma$[`deleteLog`$(\Delta R);F^{<old>}],\sigma$ [`insertLog`$(\Delta R);F^{<new>}]$).

Proposition 1 `Propagate`$(\sigma[R;F];[R,\Delta R]) \equiv$ `DiffSelect`$[\Delta R;F]$.

The formal proof of this proposition is omitted here. Readers may refer to [66] for further detail. This proposition shows that Definition 6 gives an implementation of `DiffSelect` which does not reference the base relation, and which can be implemented by one pass over ΔR, determining for each tuple of ΔR whether it induces an insertion, deletion, or modification to $\sigma[R;F]$.

When $|R| \geq |\Delta R|$, it is cheaper to re-evaluate the query expression $\sigma_P(R)$ by using the differential select operator `DiffSelect`$(\Delta R; P)$ rather than recomputing the expression

$\sigma_P(R)$ from scratch.

4.3.2.2 The Differential Project Operator

Definition 7 (Differential project)

Let R be a relation (base or derived) of scheme $\mathcal{R} = (A_1, \ldots, A_n)$. Let $\mathbf{X} = \{A_{i_1}, \ldots, A_{i_k}\}$ denotes a subset of the attributes in $\mathcal{R}$, $\texttt{tid} \in \mathbf{X}$, and $i_j \in \{1, \ldots, n\}$ $(1 \leq j \leq k)$. Let P denote the predicate $\bigvee_{j=1}^{k}(A_{i_j}^{<old>} \neq A_{i_j}^{<new>})$ if R is a (persistent) base relation; otherwise P denote the truth value `true`. The differential project operator `DiffProj` is defined as follows:

$$\texttt{DiffProj}[\Delta R;\mathbf{X}] = \sigma_P(\Pi[\Delta R;\ \texttt{old}(\mathbf{X}) \bigcup \texttt{new}(\mathbf{X})]).$$

The following proposition shows that `DiffProj` is an efficient differential form that can be used to implement $\texttt{Propagate}(\Pi[R;\mathrm{X}];(R, \Delta R))$, because they are functionally equivalent, `DiffProj` runs one pass over ΔR, and the `Propagate` runs over R' which tends to be much larger than ΔR.

Proposition 2 $\texttt{Propagate}(\Pi[R;\mathbf{X}];\ (R, \Delta R)) \equiv \texttt{DiffProj}[\Delta R;\ \texttt{old}(\mathbf{X}) \bigcup \texttt{new}(\mathbf{X})]$.

Similarly, the formal proof of this proposition is omitted here.

Example 7 Consider the following two queries over the relation `Stocks` in Example 4:

$$\mathrm{Q}_1 = \Pi[\texttt{Stocks};\ \texttt{Name}].$$

$$\mathrm{Q}_2 = \Pi[\sigma_{\texttt{price>120}}(\texttt{Stocks});\ \texttt{Name}].$$

Let $\texttt{HighStocks} = \sigma(\texttt{Stocks};\ \texttt{Price>120})$. By Definition 5, the evaluation of $\texttt{Propagate}(\sigma_{\texttt{price>120}}(\texttt{Stocks});\ [\texttt{Stocks},\Delta\texttt{Stocks}])$ requires as the inputs both the base relation `Stocks` before the updates and the base relation after the updates, which is `Stocks'`. According to Proposition 2 and Definition 7, it would be more efficient to process both queries (Q_1 and Q_2) using the differential operator `DiffProj` than directly using the definition of `Propagate`.

- Propagate(Π[Stocks; Name]; [Stocks,ΔStocks])

  = DiffProj[ΔStocks; Name].

tid$^{<old>}$	Name$^{<old>}$	tid$^{<new>}$	Name$^{<new>}$
-	-	101088	MAC
092394	OLI	-	-

- Propagate(Π[HighStocks; old(Name)∪new(Name)]; [HighStocks,ΔHighStocks])

  = DiffProj[ΔHighStocks; old(Name)∪new(Name)]

  = DiffProj[Propagate(HighStocks;[Stocks,ΔStocks]); old(Name)∪new(Name)]

  = DiffProj[DiffSelect(ΔStocks); old(Name)∪new(Name)].

tid$^{<old>}$	Name$^{<old>}$	tid$^{<new>}$	Name$^{<new>}$
120992	DEC	120992	DEC
092394	OLI	-	-

4.3.2.3 The Differential Join Operator

We first consider the join over two relations R_1 and R_2 (i.e., $n = 2$). In this case changes to the resulting relation can be induced by changes to either input operand or both of them. Therefore, the differential join operator DiffJoin should consider the following three cases when computing the total changes to the result relation of $R_1 \bowtie R_2$: (i)only R_1 changes; (ii)only R_2 changes; and (iii)both R_1 and R_2 change.

Definition 8 (Differential join)

Let P_{join} be a predicate on $\mathcal{R}_1$ and $\mathcal{R}_2$, and let $\texttt{tid}_1 \in \mathcal{R}_1$ and $\texttt{tid}_2 \in \mathcal{R}_2$ respectively. Let $P_{join}^{<old>}$ and $P_{join}^{<new>}$ denote the predicates obtained from P_{join} by attaching each attribute name with superscript $< old >$ and $< new >$ respectively. Let $P(tid_1, tid_2)$ denote the predicate "$(tid_1^{<old>} = tid_1^{<new>})$ AND $(tid_2^{<old>} = tid_2^{<new>})$". Let $\mathcal{R}$ denote $\mathcal{R}_1 \cup \mathcal{R}_2$. We define the operator DiffJoin as follows:

$$\texttt{DiffJoin}_{P(tid_1,tid_2)}(R_1, R_2)$$
$$= \texttt{DJoin}_{P_{join}}(\Delta R_1, R_2) \bigcup \texttt{DJoin}_{P_{join}}(R_1, \Delta R_2) \bigcup \texttt{DJoin}_{P_{join}}(\Delta R_1, \Delta R_2).$$

where

- $\texttt{DJoin}_{P_{join}}(\Delta R_1, R_2)$

  $= \texttt{Outerjoin}_{P(tid_1,tid_2)}(\texttt{attach}[\bowtie_{P_{join}^{<old>}}(\texttt{deletions}(\Delta R_1),\ R_2);\ \texttt{old}(\mathcal{R})],$

  $\texttt{attach}[\bowtie_{P_{join}^{<new>}}(\texttt{insertions}(\Delta R_1),\ R_2);\ \texttt{new}(\mathcal{R})]).$

- $\texttt{DJoin}_{P_{join}}(R_1, \Delta R_2)$

  $= \texttt{Outerjoin}_{P(tid_1,tid_2)}(\texttt{attach}[\bowtie_{P_{join}^{<old>}}(R_1,\ \texttt{deletions}(\Delta R_2);\ \texttt{old}(\mathcal{R})],$

  $\texttt{attach}[\bowtie_{P_{join}^{<new>}}(R_1,\texttt{insertions}(\Delta R_2);\ \texttt{new}(\mathcal{R})]).$

- $\texttt{DJoin}_{P_{join}}(\Delta R_1, \Delta R_2)$

  $= \texttt{Outerjoin}_{P(tid_1,tid_2)}(\bowtie_{P_{join}^{<old>}}(\texttt{deleteLog}(\Delta R_1),\ \texttt{deleteLog}(\Delta R_2)),$

  $\bowtie_{P_{join}^{<new>}}(\texttt{insertLog}(\Delta R_1),\ \texttt{insertLog}(\Delta R_2)).$

Proposition 3 Let the differential substitution list SL be $\{[R_1, \Delta R_1],[R_2, \Delta R_2]\}$.

$$\texttt{Propagate}(\bowtie_{P_{join}}(R_1, R_2);\ \{[R_1, \Delta R_1],[R_2, \Delta R_2]\}) \equiv \texttt{DiffJoin}_{P(tid_1,tid_2)}(R_1, R_2).$$

This development can be generalized to the join of an arbitrary number of base relations. Let $Q = R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_n$ $(n \geq 2)$ denote an arbitrary join expression. The equation $\texttt{Propagate}(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n;\ \text{SL}) \equiv \texttt{DiffJoin}_{P_{tid_1,\ldots,tid_n}}(R_1, R_2, \ldots, R_n)$ holds in general. Also, when the number k $(k \leq n)$ relations have been changed since the last execution of Q, to evaluate

$\texttt{DiffJoin}_{P_{tid_1,\ldots,tid_n}}(R_1, R_2, \ldots, R_n)$, we need to consider only $2^k - 1$ cases, each representing one type of change effects. The total changes to the result of Q will be the union of these cases. According to the associative and symmetric property of relational join, we may assume that the first k relations (i.e., $R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_k$, $k \leq n$) are those that have been changed without loss of generality.

By associating a truth table T, with k columns and $p = 2^k - 1$ rows, to the query Q, it is easy to compute each possible combination of joins, which needs to be considered when computing the changes to the result of Q after k operand relations have been changed. Each column of the T table corresponds to an updated relation in Q since the last execution of Q. Each row represents one possible case that the computation of $\texttt{DiffJoin}(R_1 \bowtie R_2 \bowtie$

$, \ldots, \bowtie R_n$) considers. The formal proof of this proposition is also omitted here. Readers may refer to [66] for detail.

Example 8 For a query Q = $R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_n$ ($n \geq 2$), consider the case when $k = 3$ and $k < n$. Based on the T table associated with Q below, we need to consider only seven cases: (1)R_1, R_2, R_3 all change; (2)only R_1, R_2 change; (3)only R_1, R_3 change; (4)only R_2, R_3 change; (5)only R_1 changes; (6)only R_2 changes; (7)only R_3 changes. Each row in the T table corresponds to one possible combination of join required to compute the changes to the result of Q.

T1	T2	T3	DiffJoin$(R_1, R_2, R_3, \ldots, R_n)$ =
1	0	0	DJoin$(\Delta R_1, R_2, R_3, \ldots, R_n)$
0	1	0	$\cup$ DJoin$(R_1, \Delta R_2, R_3, \ldots, R_n)$
0	0	1	$\cup$ DJoin$(R_1, R_2, \Delta R_3, \ldots, R_n)$
1	1	0	$\cup$ DJoin$(\Delta R_1, \Delta R_2, R_3, \ldots, R_n)$
1	0	1	$\cup$ DJoin$(\Delta R_1, R_2, \Delta R_3, \ldots, R_n)$
0	1	1	$\cup$ DJoin$(R_1, \Delta R_2, \Delta R_3, \ldots, R_n)$
1	1	1	$\cup$ DJoin$(\Delta R_1, \Delta R_2, \Delta R_3, \ldots, R_n)$

Let $\mathcal{R}_i$ be the scheme of relation R_i ($1 \leq i \leq 3$) and $\mathcal{R}$ denote $\cup_{i=1}^{n} \mathcal{R}_i$. The definitions for the above DJoins are similar to those in Definition 8. For instance,

DJoin$_{P_{join}}(\Delta R_1, R_2, R_3, \ldots, R_n)$

= Outerjoin$_{tid}$(attach[deletions$(\Delta R_1) \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n)$; old$(\mathcal{R})$],

attach[insertions$(\Delta R_1) \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n)$; new$(\mathcal{R})$]).

DJoin$_{P_{join}}(\Delta R_1, R_2, \Delta R_3, \ldots, R_n)$

= Outerjoin$_{tid}$(attach[deletions$(\Delta R_1) \bowtie R_2 \bowtie$deletions$(\Delta R_3) \bowtie \ldots \bowtie R_n)$; old$(\mathcal{R})$],

attach[insertions$(\Delta R_1) \bowtie R_2 \bowtie$insertions$(\Delta R_3) \bowtie \ldots \bowtie R_n)$; new$(\mathcal{R})$]).

Observe that the above example exhibits an interesting optimization problem, namely, the efficient execution of a set of n-ary join expressions in which intermediate results can be

reused among several *SPJ* expressions. For instance, when $n > 4$ in the above query Q, let $W_1 = R_4 \bowtie \ldots \bowtie R_n$ and $W_2 = R_2 \bowtie R_3 \bowtie W_1$. Saving W_1 and W_2 as intermediate results, and then re-using them in the evaluation of each of the seven `DJoin` expressions above, we may easily speed up the processing of `DiffJoin`. This mechanism works effectively when n is larger than k.

The idea of using the truth table to facilitate the combination of possible joins was borrowed from the research in updating materialized view [14, 44]. We minimize the cost of constructing such a table by using as the columns of the table only the number of changed relations, instead of the n operand relations in the query expression $\Pi_X(\sigma_F(R_1 \bowtie R_2 \bowtie , \ldots, \bowtie R_n))$.

Moreover, in the DIOM system, we apply several conventional query optimization techniques used in both centralized and distributed environment to further reduce the cost of `DiffJoin` operation. For example, given a continual query Q, denoted by $\Pi_X(\sigma_F(R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_{50}))$, which request access to 50 classes/relations from 20 different information sources. Assume only two relations (say R_1 and R_2) have been updated since the last execution of the query Q. By using the commutativity of selection and projection over joins and associativity of joins, this query Q will first be decomposed into the following two subqueries: $SubQ_1 = \Pi_{X_1}(\sigma_{F_1}(R_3 \bowtie R_4 \bowtie, \ldots, \bowtie R_{50}))$ and $SubQ_2 = \Pi_{X_2}(\sigma_{F_2}(R_1 \bowtie R_2 \bowtie Result(SubQ_1)))$, where $X = X_1 \cup X_2$ and F_1 is a selection condition over $R_3, R_4, \ldots, R_{50}$ and F_2 is a selection condition over R_1, R_2, and the result of $SubQ_1$. Now the evaluation of the `DiffJoin` operator over Q is reduced to `DiffJoin` over the subquery $SubQ_2$. The system performance for processing this CQ will be greatly improved, because (i)the evaluation of $SubQ_1$ can be done directly against the previous execution result of Q cached at the client side, and (ii)comparing with the original query Q, the size of $R_1 \bowtie R_2 \bowtie Result(SubQ_1)$ is much smaller both in cardinality and in degree than $(R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_{50})$. This approach is particularly beneficial when the selectivity of F_1 is high and the project list X_1 is small.

4.3.3 The Differential Re-evaluation Algorithm

We now outline an algorithm for re-evaluating continual queries (limited to SPJ expressions) differentially.

Algorithm 1 (The DRA algorithm)

Input:

- the SPJ definition of the continual query Q, i.e., Q = $\Pi_X(\sigma_F(R_1 \bowtie R_2 \bowtie, \ldots, \bowtie R_n))$, where X denotes the projection list and F denotes the selection predicate over $R_1, \ldots, R_n$;
- the contents of the base relations R_i $(1 \leq i \leq n)$ after the last execution of the CQ;
- the differential relations ΔR_i $(1 \leq i \leq n)$;
- the timestamp of the last execution of this CQ, say E_i;
- the complete set of the result produced by the last execution of the CQ.

Output: the result of the current execution of the query Q.

Procedural Steps:

1. Build the truth table T with k columns $(k \leq n)$ and p rows, $p = 2^n$. Each column is corresponding to a relation in the SPJ expression, which has been changed since the last execution of Q.
2. For each row i $(1 \leq i \leq p)$ of the table T, construct the associated SPJ expression, by substituting R_i in Q with ΔR_i when the binary variable $T_{ij} = 1$. For each of these SPJ expression, denoted by G = $S_1 \bowtie S_2 \bowtie, \ldots, \bowtie S_n$, evaluate G by its differential form `DJoin`$(S_1 \bowtie S_2 \bowtie, \ldots, \bowtie S_n)$.
3. Perform the union of the results obtained from each computation in Step 2.
4. Based on the epsilon specification of the CQ, assemble the final set of the result to be returned to the users.

For example, let ΔR_Q denote the result generated by Step 3.

- If the user wishes to see only the differential result since the last execution of the Q, say $\mathrm{E}_i(\mathrm{Q}, t_i)$, without deletion notification, the result to be returned can be computed by

  $\sigma_{timestamp>t_i}(\texttt{insertions}(\Delta R_Q))$.

- If the user needs to see the complete set of the result matching the query, we return $\mathrm{E}_i(\mathrm{Q},t_i) \bigcup \sigma_{timestamp>t_i}(\texttt{insertions}(\Delta R_Q))$.

- If the user wants to be notified all the deleted tuples since the last execution of the CQ, we simply compute the $\sigma_{timestamp>t_i}(\texttt{deletions}(\Delta R_Q))$.

We use the relational model to describe the DRA algorithm. This is a design choice. In principle, CQs could be written in query languages that assume other data models. In this chapter, we do not address the issue of query language translation, which by itself is a significant research topic. The extension of the DRA algorithm to object-oriented models will become important when more data become available on object-oriented databases, and more queries are written in object-oriented query languages. At present, most of organized data are stored in relational database management systems and queries written in SQL. Similarly, most of unorganized data (such as WWW pages) are stored in files and queries over these data are submitted through simple GUIs and can be translated into SQL.

4.4 Processing Continual Queries: Simple Examples

In this section, we use examples to illustrate the differential evaluation based on differential relations (log data) for processing continual queries. We also compare our approach with the timestamp-based transformation strategy [102], and demonstrate the benefits of using differential evaluation strategy in a database environment WHERE data items can be appended, removed, or modified dynamically.

Example 9 Suppose the user wants to install the following two queries as continual queries:

```
Q1:  SELECT * FROM Stocks
```

```
        WHERE price < 120

Q2:  SELECT * FROM Stocks
     WHERE name = "DEC" OR name = "MAC"
```

(1) Using the differential evaluation based on differential relations, we may easily transform the above queries into the queries over the differential relation ΔStocks as follows:

```
IQ1:  SELECT * FROM ΔStocks WHERE price < 120

IQ2:  SELECT * FROM ΔStocks WHERE name = "DEC" OR name = "MAC"
```

The initial execution of Q1 returns (101088, USL, 100). The initial execution of Q2 returns (120992, DEC, 150). After the initial execution of a continual query, say Q1 or Q2, the subsequent executions of the same query will be carried out by the differential query IQ1 or IQ2. Suppose now the database is changed by the transaction T described in Example 4. According to the Stocks' relation given in Example 5, two tuples (101088, USL, 100) and (101088, MAC, 117) are qualified for Q1. The first one remains the same and the second one is a new tuple inserted by T. The execution of the differential query IQ1 guarantees that only the newly added tuple (101088, MAC, 117) is returned. When the base relation is large in size, using the differential query evaluation based on differential relations will drastically reduce the computation cost of the continual queries. Similarly, executing Q2 as a continual query after transaction T is committed, the tuples (120992, DEC, 149) and (101088, MAC, 117) will be returned by IQ2.

(2) Using the timestamp based transformation approach proposed by [102], two preconditions must be satisfied:

- The database is restricted to append only in the sense that data items are appended to the database as they arrive and are never removed or modified [102].
- Each object instance (relation tuple) should have a timestamp associated with to represent its creation time.

The queries `Q1` and `Q2` will first be transformed to time-stamped queries by adding timestamp related conditions to the query expression, for example:

```
MQ1: SELECT * FROM Stocks p
     WHERE price < 120 AND p.st < CURRENT_TIMESTAMP

MQ2: SELECT * FROM Stocks p
     WHERE name = "DEC" OR name = "MAC"
     AND   p.st < CURRENT_TIMESTAMP
```

If the system may provide the timestamp, say `d`, of the previous execution of a continual query, then the above time-stamped queries may be further rewritten to reduce the search space.

```
MQ1: SELECT * FROM Stocks p
     WHERE price < 120 AND p.st > d

MQ2: SELECT * FROM Stocks p
     WHERE name = "DEC" OR name = "MAC"
     AND   p.st > d
```

In our view, adding timestamp to every tuple of the existing legacy systems is far more expensive than maintaining differential relations, especially when the database is large in size. Also many real-world applications, for which continual queries would be a useful tool, allow data items to be appended, removed or modified at any time.

4.5 Performance Evaluation of DRA

The performance of the *Differential Re-evaluation Algorithm* (incrementally computing query results) is compared with the *Brute-Force Algorithm* (re-computing query results) in this section. These two algorithms are called DRA and BFA, respectively in the results shown below. All experiments are conducted in a local area network, with the effect of remote fetching data minimized. The stock data used in the experiments are collected

from Yahoo! Finance Web site (http://finance.yahoo.com). Queries in the experiments are *Selection Only*.

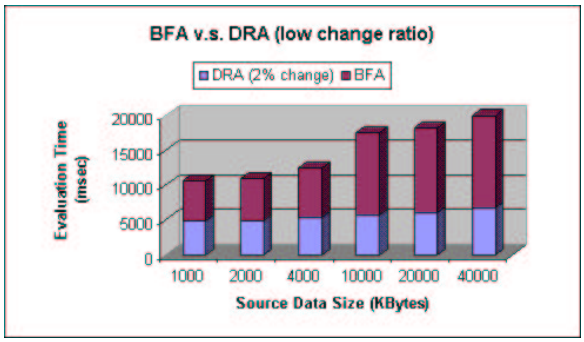

Figure 35: BFA v.s. DRA under fixed low change ratio

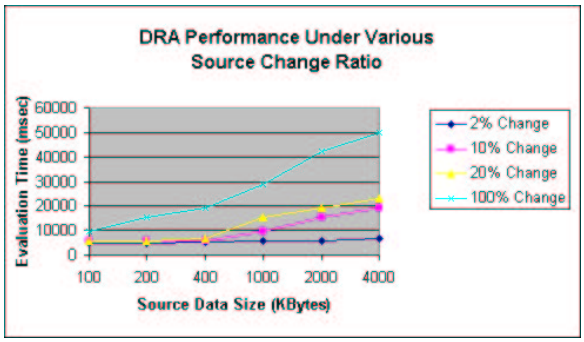

Figure 36: DRA performance with varying source change ratio

Figure 35 and Figure 36 demonstrate the benefit and cost of using the DRA algorithm for continual query evaluation. From Figure 35, we can see that when the source data does not change much (in this case 2%), it is beneficial to use DRA comparing to BFA. And the larger the data source, the more savings can DRA get, as long as the source change ratio is kept at low level. Figure 36 shows the execution time of a continual query using DRA when the source change ratio is higher. In this case, using DRA to evaluate continual queries does not offset the cost, although the cost factor is exaggerated because all the data sources are accessed via local area network instead of remotely whereas DRA usually makes use of a local differential relation for evaluation.

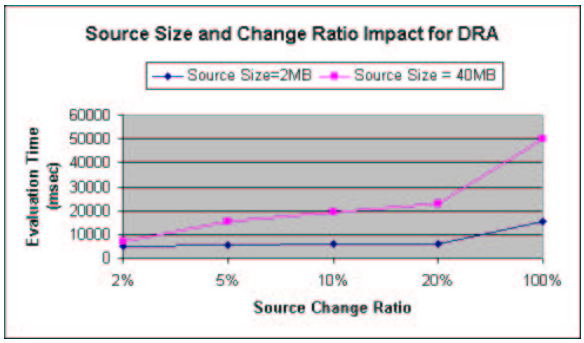

Figure 37: Source size and change ratio impact on DRA

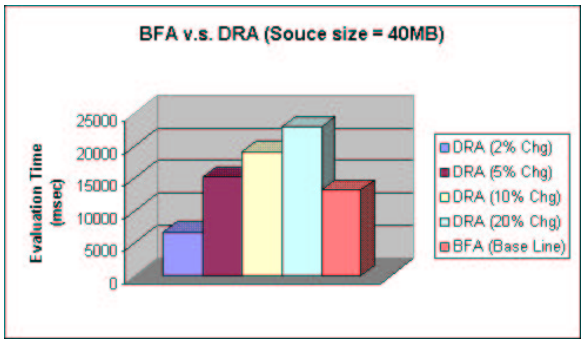

Figure 38: DRA performance with fixed source size and varying change ratio

Figure 37 and Figure 38 illustrate the impact of different parameters on the DRA algorithm, namely *source change ratio* and *source data size*. From Figure 37 we can see that when data size is small (i.e., 2MB), the effect of data source change ratio is not huge, while it is not the case when source data is large. Figure 38 compares the performance of BFA and DRA with fixed source data size. When the change percentage is high, DRA does not win at all. Again, this argument is based on the fact that we filtered out the network cost. It would be interesting to see how adding network cost into the picture will change the result.

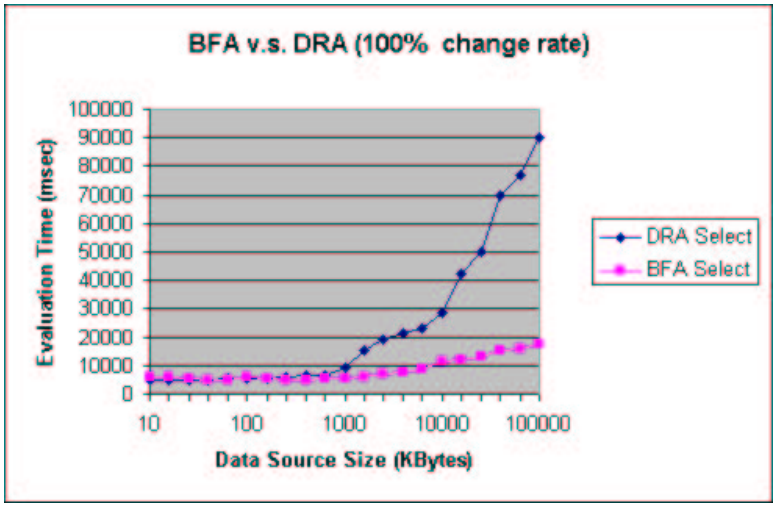

Figure 39: Differential v.s. Brute-force query evaluation (100% source change)

For an extreme case of 100% source changes, the performance result is shown in Figure 39. We can see the relative flat curve for evaluating CQs using the brute-force algorithm and the exponential time increase of the DRA approach. This is because differential query operators are more expensive to evaluate (because of self-join). Although this may not be obvious when source data size is small (<1MB in the figure), we can see the two curves diverge quickly when source data size is larger than 1MB. Therefore, combining the figures demonstrated above, we can conclude the DRA algorithm only benefits when source data change is small. Otherwise, the amount of data shipping and costly differential query operators will override the benefits.

4.6 Discussion

In previous sections we have defined the differential forms for the three basic relational algebraic operations: `Select`, `Project`, and `Join`, and developed a differential re-evaluation algorithm (DRA). The key idea of the DRA method is to transform a continual query over the source data (base relations) into a differential query that runs over the corresponding delta relations. As formally proved in [66], for any SPJ expression, using differential re-evaluation method is functionally equivalent to the complete re-evaluation of the query.

In this section we discuss a number of issues related to performance optimization opportunities for continual query processing.

4.6.1 Strawman Performance Arguments

Although there is no space in this chapter for a more detailed performance analysis, we argue informally that there are many important scenarios in which DRA wins over algorithms that operate on the base data instead of results.

First of all, we observe that the overwhelming majority of queries return a table of results that is much smaller than the base data. (Otherwise, the query would be considered not selective enough and the results not particularly useful.) Therefore, DRA processing of the next query execution on top of results will be much faster, reducing both I/O and CPU requirements and communication overhead. In general, caching the results on the client side makes the servers more scalable with respect to the number of clients.

Second, since results are combined from many sources into a local table, DRA processing of results will avoid both translation from the base data to an interoperable format. Moreover, if the volume of relevant updates is smaller than the results (which is the common case), then we are further reducing the network traffic.

Third, each server only generates delta relations when communicating with the clients. This is easier for interoperation than trying directly to integrate active databases and materialized views. To the best of our knowledge, there are no practical methods for combining them in a heterogeneous environment.

On the other hand, we note some limitations of the DRA algorithm. For example, when

the results turn out to be large (poor selectivity of the query), then a lazy evaluation and transmission of results is necessary. Another important assumption of the DRA algorithm is the availability of delta relations from every information producer. This may not be trivial for legacy databases. But as we mentioned before, there is no easy way to integrate legacy active databases or materialized views, either.

4.6.2 Query Refinement

First, we should test the CQ condition based on the differential updates before every execution. If the updates occurred in between of the two consecutive executions have no impact on the previous query result set, we consider them as irrelevant updates to the continual query. Thus, no computation is performed for this CQ, because in this case, nothing needs to be returned if the user is only interested in the differential result. When the user asks for the complete answer, we simply return the result of previous execution.

In addition, for each SPJ expression in step 2 of the DRA algorithm, it is necessary to determine its execution strategy. One way to find a good execution strategy is simply to use the heuristics such as `Select` before `Join`, extracting common subexpressions, cheaper selection predicate before expensive ones). This approach might be most appropriate if one does not have access to an appropriate query optimizer. An alternative approach is to have a DBMS query optimizer generate the strategies. The differential form of a query can be regarded as a query to a database that consists of base relations and differential relations.

4.6.3 Garbage Collection of Differential Relations

As the source data changes, their differential relations grow accordingly. To keep the differential relations to a bounded size, we need to garbage collect the portions of the differential relations that are no longer useful. The technical details of the solution are beyond the scope of this paper. We outline the basic idea here. First let us consider the case of a single active CQ in the system. Each time a new query result $Q(S_i)$ is obtained, we can retire the differential relations referring to database states prior to t_i. This is intuitively easy to understand since only the data in the differential relations with the timestamps later than t_i will be needed for the processing of $Q(S_{i+1})$. Informally, we call these portions of the

differential relations "active delta zone".

With multiple active CQs in the system, the garbage collection algorithm is an extension of the basic idea outlined above. For each CQ, we define its active delta zone. For the whole system, we define the system active delta zone as the union of the active delta zones of all CQs. Assuming that each CQ will make progress, its active delta zone will move forward in time. The system active delta zone will move forward as a consequence, with its boundary delimited by the "oldest" active delta zone. All the data in the differential relations that fall outside the system active delta zone can be garbage collected, since they will not be used by any active CQ.

4.6.4 Generation of Delta Relations

It should be pointed out that, although we use the relational model terminology and concepts in the design and description of the DRA for clarity and simplicity, the DRA itself takes as input the updates from different information sources. These updates are described as differential relations in this paper, as differential relations have a very simple and clear form and content for representing updates in terms of modifications, insertions, and deletions. For the relational information source providers, the generation of different relations is quite straightforward. For those information sources other than relational databases, simple translators (as part of the DIOM services [65]) will be used to catch the updates in the form of differential relations. For example, file system updates can be captured by either operating system or middleware and translated into a differential relation and fed into DRA. This is in contrast to the conceptual difficulties in the integration of active databases and view materialization, as well as the practical difficulties of implementing these powerful database techniques in non-database environments such as file systems.

4.7 Implementation Consideration for DRA

The DRA implementation is under development in the prototype CQ system. It has not reached the working stage yet. However, some tools and utilities have already been implemented for the DRA development. The description of the available tools and future implementation plan is listed below:

- Step 1:

  Automatically generate delta tables for objects of a particular data source, which capture the updates on the source objects .

  A wrapper function has been implemented in Perl to facilitate auto-generation of delta tables for target data objects.[3]

  For example, if we want to create the delta table for the source object table "STOCK" at data source "FINANCIAL", given the structure of "STOCK" as follows:

SYMBOL	PRICE

  A delta relation with the name "DELTA_FINANCIAL_STOCK" will be created (*tid* is not necessary since it can be derived from the key attributes):

OLD_SYMBOL	OLD_PRICE	NEW_SYMBOL	NEW_PRICE	TIMESTAMP

  This delta table will record all the updates on the stock objects performed locally at the data source. The CQ server will make use of this delta table to execute differential queries on "STOCK" object.

- Step 2:

  Given a continual query $(Q, T_{cq}$, `Stop`$)$, transform the query component Q into an algebraic query graph with selection, projection, join operators as the internal nodes and data source objects as the leaf nodes.

- Step 3:

  Design and implement delta query operator for each primitive algebraic operations such as $DiffSelect$, $DiffProj$, and $DiffJoin$.

- Step 4:

  Design and implement a generic delta query transformation module which maps an arbitrary query in algebraic expression into a delta query expression.

[3]The data objects from the target data source are first mapped to the continual query system object model, which is a relational model as in RDBMSs.

- Step 5:

  Design and implement the refresh strategies to keep the delta relations up to date with the state of data sources. There will be a system component (we call it *Refresher*) running continually in background, which is independent of the continual query component. Basically, the *Refresher* will insert into the delta relations the new updates at the data source.

- Step 6:

  Design and implement a garbage collection module for delta relations. Since the delta relations grow whenever the *Refresher* updates them. Consequently, delta relations may become larger than the base tables. As a matter of fact, the objects in the delta table may not be useful any more when there is no continual query in the system that is using them(recall Section 4.6.3). Thus we have implemented a *Garbage Collector* which removes those delta objects that are out of date. More concretely, when the timestamp T_1 of a record r_1 in the delta relation DELTA_SRC_OBJ is older than the last execution timestamp of a particular *continual query*, it means that the corresponding update of the source object must have been accounted for during the last execution of the *continual query*. Therefore, record r_1 is considered "garbage" for the particular *continual query*. When T_1 is older than all the timestamps of the registered continual queries related with the object SRC.OBJ, record r_1 can be deleted from the delta relation. The *Garbage Collector* is also scheduled to be running periodically.

CHAPTER V

CHANGE RESPONSE FOR INFORMATION EXCHANGES

Web page monitoring tools [16, 30, 21, 49, 68, 69, 105, 106, 110] have been developed and used during the last few years to help people keep current as information on the Web has changed. These monitoring systems differ in quality of difference generation and summarization as well as methods of notification delivery. To date, the uses have been primarily geared towards keeping people up-to-date through various notification mechanisms such as email and more recently using RDF Site Summary (RSS) [89] as a mechanism to report changes to Web content. However, very little work has been done to explore the use of such tools for automating and integrating change response to business processes impacted by the changes.

Business information exchanges often involve complex and often manual processes. Take retailing business for an example, a retailer usually exchange contracts, purchasing orders, bills, invoices, and shipment information with its supplier. There could be occasions when something goes wrong (contract changes, shipment delay, etc.). Resolving these problems is most often a manual process, relying heavily on phone, fax, or email. Both the retailer and supplier might suffer losses if the problems are not resolved promptly.

Because many of these updates will trigger certain "status" changes, the ability to monitor the changes and respond to them becomes inevitably important. The problem gets aggravated given the fact that a lot of the updates are posted in different formats and on various dates. Therefore, the ability to formulate a response based on the type, semantics, and value of the change is an important step towards automating the monitoring and change response process.

This chapter explores only several of a literally unlimited number and type of responses that could be formulated. In particular, we focus on providing semantic and recommended

actions as part of the human notification system. In addition, we look at some simple system changes that can be made to improve the overall information exchange ecosystem. The proposed framework *BizCQ*is discussed in more details in [101]. This framework is an extension of the current WebCQ system implementation [109]. The research work presented in this chapter is still ongoing.

5.1 Applied Monitoring

Today's information exchanges are composed of many moving parts. Figure 40 shows a 'typical' flow of the components involved in an integrated information exchange between two business partners.

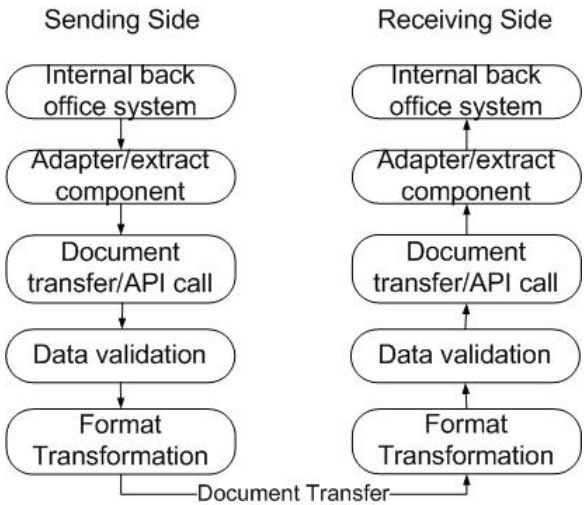

Figure 40: Information Exchange Between Business Partners

The figure represents a very simplified view of the components but will suffice for purposes of exploring some of the typical human responses to changes required by an external business partner. During this process, the problem is not isolated to computer-to-computer communication. The human element is also often included in the information that is published by an external partner. This information can include such things as:

- Contact information
- Shipping information

- Compliance/legal information
- Procedural information

The impact of these changes depends on their categories and type s. For example, a shipping address change can cause major delays as well as additional charges if it is not caught and remedied in time. Additionally, many companies require monetary restitution for non-compliance with published specifications. These 'charge backs' can be quite expensive if the problem is not noticed and remedied immediately.

Based on to the variety of both formats and types of information that must be monitored, this task typically consumes a large portion of an individual or individuals' time during the normal course of business. This consists of manually visiting the various business partner Web sites and attempting to visually determine what has changed and how that change may impact the business. This is not only time consuming but also tedious. In fact, several companies we have interviewed discussed just how manual this process is. In some cases, a 3-ring binder is kept with printed copies of the sections of the business partner's Web site. This printed copy is then visually reconciled against a current site to attempt to ascertain what information has changed. The BizCQ framework is aiming at reducing this manual process by providing system support and a set of toolkits in information monitoring and change management.

In the set of experiments we conducted, the average time for detecting changes for each of the sentinels was 1.229 seconds. It typically takes between 2 minutes and 30 minutes for a human to determine this same information. The variance is due the human's lack of information to quickly compute the difference between the two versions. The cumulative effect of this time savings can be considerable considering a single company may employ as many as 2000 sentinels.

5.2 Challenges in Change Management

The 'change management' process consists of three broad steps: tracking the changes made by business partners, deciding what needs to be done for each change (who needs to be informed, what steps need to be taken), and implementing the change response.

As a first step, there is no standard way of figuring out whether a business partner wants to make a change to their eBusiness connection. For larger companies this step is fairly simple: they get an email or phone message from their partners informing them of any changes they have decided upon. For smaller partners, especially in the retail apparel space, the process is much more cumbersome and time consuming. The supplier is expected to scan the Web sites regularly and react promptly to the changes to avoid punitive 'charge backs' or expense offsets.

Once the change has been detected, appropriate people within the organization need to be notified outside of the IT department. For routing changes, it may be the logistics coordinator or warehouse manager. For changes in vendor guidelines, it may well be someone in accounting.

Implementing the change can be as simple as updating contact information in a database or as complex as map transformations. Since many companies keep much of their business partner information in paper form, much of their time is spent in wading through paper to figure out the implications of each change and deciding the next steps. For example, a typical transaction format change involves:

1. Locating the details on partner formats in paper manuals.
2. Determining the implications of the change.
3. Making contact with the eBusiness coordinator of the business partner.
4. Scheduling time for discussions.
5. Implementing the map change.
6. Testing.

Not only is this process very complicated, but it needs to be repeated for scores, if not hundreds of partners, each with its own unique set of rules. So change management often saps up expensive resources in performing repetitive, low value-added tasks.

A lack of standards in communication between business partners, multiple users of information, use of multiple types of databases including paper manuals to record partner

information, and constant updates all make tracking all information on business partners a major challenge.

There is also fragmentation at every level of B2B: the interconnection protocols, the data formats and the business processes. And given that companies typically deal with hundreds, often thousands of partners, this investment increases exponentially.

In many cases, much of the information is still on paper. In some companies, voluminous data like partner profiles, policies, etc. are in paper manuals and wading through these to access and/or update the information is a major challenge.

5.3 Assisting Human Responses

After the discovery of a change, the next task of any system (human or otherwise) is to classify the type of information, the relevance, and the impact.

To help the system provide relevance, we have proposed an extension to the WebCQ system to provide annotation capability to the definition of sentinels. This annotated information provides hints to both the human respondent as well as the automated response system. Currently proposed annotations include:

- Information type (e.g. contact information)
- Priority
- Impact area (e.g. translation impact)
- Recommended action
- Key words

These annotations allow the creator of the sentinel to pass some elements of their knowledge of the information and its potential impact directly into the system and thus onto any other constituents that may make use of the change monitoring.

5.4 Enacting Appropriate Change Response

The current online WebCQ system [109] utilizes email-based notifications to alert users of changes. The system maintains an archive of the information requested by the registered

sentinels and in this way can provide a valid difference engine and can create a visual representation of the changes.

This visual representation includes highlight marks indicating the changed content. In addition to this information, the system can be extended to include the annotations as provided by the creator of the sentinel. In this way, a change notification can be routed to the most likely impacted system and an appropriate response can be formulated.

According to the sample set that we worked with, the most common responses categorized by change type are shown in Table 4.

Table 4: Change Response Categories

Change Class	Domain Change Category	Actual Change	Typical Response
Structural	N/A	Added a new heading/image	No action
Presentational	N/A	Added a new image	No action
Informational	Update contact list	Added a new technical contact	Print out new contact information, include printout in a 3-ring binder.
Semantic	Update shipping information	Distribution center for some stores has changed	Contact shipping department, inform them of changes.
Semantic	Update a transaction format or requirement	Transaction format for purchase orders has changed: previously optional field is now mandatory, new optional field is added	Make programmatic change to translator map. Test and move change to production.

Each of the types of changes depicted above elicits a different type of response from the receiver of the notification. It is our intent to explore ways to both help the human response and, where possible, automate that response.

5.5 Automated Change Response

For purposes of automating the response formulation, it is necessary to isolate and evaluate both the qualitative and quantitative measure for each change. This process, normally accomplished by the human reader, must at some level be done at the machine level in order to determine an appropriate response.

The primary extensions to WebCQ include the ability to notify a system via a Web service call, the ability to include annotation information throughout the system, and the ability to categorize the qualitative and quantitative changes.

Previous versions of monitoring systems such as WebCQ have been created with the intended audience being human. This has generally been handled via standard email messages. The work being presented here has a different intended recipient. The final recipient very well may be a human, but there are potential intermediate steps that may automatically take place based on a set of rules and conditions.

Initially, the set of automated tasks that can be accomplished will be a small subset of the overall response that must ultimately be made to support change requests. However, it is conceivable that this set of tasks grows both in complexity as well as breadth to cover a good portion of the mundane change responses that are normally handled by humans.

The Web service notification allows the system to dynamically change and grow its capabilities over time. Through this loosely coupled connection, the change monitoring, the change measurement, and the change response can all be modified and expanded without impacting the other subsystems.

The change response service is responsible for determining the appropriate response based on the information passed in from the change detection system.

It is the responsibility of the change response services to take action based on this information. The set of actions could be any of this set:

- Create/deliver human notification
- Lookup additional information
- Create change to a transaction map
- Create change to a translation rule
- Flag a transaction set for possible errors
- Log a message
- Forward a request to an external handler

Any number of these tasks could be triggered due to a notification from the change monitoring system. The appropriate response will depend on a number of things including:

- Type of change
- Value of change
- Complexity of change
- Necessary response
- Intrusiveness of response
- Operator comfort with automated change
- Specific rules associated with the change

A prototype has been built for testing these ideas [101]. It is based on a rudimentary set of rules, but it is conceivable that a full strength rules engine and feedback system could be implemented to handle specialized change response tasks.

CHAPTER VI

RELATED WORK

Terry et al [102] proposed continuous queries for monitoring information change. However, their proposal made several assumptions that seriously restricted the applicability of their technique to the Internet. Perhaps the most significant assumption is the limitation of database updates to append-only, disallowing deletions and modifications. Since this assumption is used in their query transformation algorithm, it has been difficult to relax it [13], when following their definition of continuous queries. This is one of the motivations for our new definition in Section 2.1 under a new name, *continual queries*.

6.1 Active Databases and Materialized Views

Active Databases

Most of active database systems [115] provide facilities [19, 73, 91, 41, 97, 40] that allow users to specify, in the form of rules and actions to be performed following changes of database state. These systems support powerful rules and allow general events, conditions, and actions, and therefore are difficult to implement efficiently. The result is widely adopted in a restricted form of implementations (e.g., built-in triggers in relational database management systems such as Oracle, Sybase, and Informix). Active query, introduced in Alert [91], is yet another form of ECA rules, similar to continuous queries [102], and usually assumed to work in an append-only environment. Active queries are more sophisticated than triggers, since they can be defined on multiple tables, on views, and can be nested within other active queries. However, active queries rely heavily on a number of extensions specific to the IBM Starburst DBMS [40]. Like continuous queries [102], the append-only assumption seriously restricts the applicability of active queries to the Internet environment where data is appended, removed, or replaced constantly.

More generally, active databases [115] support rule monitoring, triggers, alerters, and

even more powerful mechanisms. Active databases support user-specified refresh policies, in contrary to work on materialized views; but to accomplish this, mechanisms like triggers and alerters need to be available in the relevant databases. In comparison to this, our work does not require each component database to be armed with such facilities.

Our work will make use of active database support, if available. But for large scale information monitoring in the Internet we cannot count on their availability. Our work can be seen as active support in an inter-operable environment.

Although recent active database research has been working on increasing triggering capabilities [43], the restriction in the architectural design makes it difficult to scale to large scale, such as providing triggering and notification services for the Internet/WWW. In OpenCQ/WebCQ systems [62, 64, 68, 69], we are aiming at large scale information monitoring by employing general mediator/wrapper architecture, which provides adaptive system scalability.

Materialized Views

Materialized views store a snapshot of selected database state. When a database is updated, the materialized view must be refreshed to reflect the updates. A naive solution is to re-materialize the view from the base data. In contrast, incremental update algorithms are believed to carry lower execution cost if changes to the database are moderate [44, 59]. Three approaches have been described previously. The first approach refreshes the view immediately after each update to the base table [14]. The second defers the refresh until a query is issued against the view [88]. The third refreshes the view periodically [59]. The main tradeoff in choosing among these approaches is the staleness of the view data vs. the cost of updating it. Most of the algorithms in the literature [14, 59, 44, 52, 38] work in a centralized database environment, in which the materialized view and its base tables co-reside. The study on distributed materialized view management has been primarily focused on determining the optimal refresh sources and timing for multiple views defined on the same base data [93, 92]. Other works on distributed environments include quasi-copies for replication [5] and update anomalies in data warehouses [124]

Materialized views are basically stored continual queries that get re-evaluated whenever

the views are updated against the base tables. Different materialized view maintenance algorithms differ in their concrete ways to compute deltas. They also differ in query languages and data models, as well as delta-relation formats.

The Differential Re-evaluation Algorithm is different from the view maintenance algorithms in materialized views in that DRA is a query refresh approach while view maintenance algorithms are base-table refresh algorithms. CQ system maintains *delta relations* [60] to incrementally maintain changes to data. The *delta relations* reside at the CQ server side when source-side delta relations are unavailable.

Compared to systems supporting materialized views, CQ systems run outside data sources that may or may not support ECA rules, or views. A continual query is only indirectly coupled to the data sources it monitors, through CQ triggers.

6.2 Structured/Semi-structured Information Monitoring

The concept of continual queries was motivated by the increasing demand on event-driven information delivery. The work on continuous queries in append-only databases by Terry et al [102] started this line of research. The design of CQ systems has been originally influenced by the work in active databases, view materialization (discuss in Section 6.1, and general matching networks in production systems.

A set of matching network structures [34, 74, 42, 54]. have been proposed in AI production systems research. The unique data structures are used for efficient matching and evaluating predicates in the rule conditions. In CQ systems, due to complicate data change monitoring requirements and specific architectural design, we cannot apply these techniques directly. System specific structures are designed to work with other system components in evaluating continual queries. Our work is also related to general publish/subscribe systems and filtering services [4, 31, 7, 119], in which efficient matching and filtering algorithms are crucial.

Traditional multi-query optimization techniques [94, 95] are not practical in a continual query systems because of its open architecture, the scale of the system, and unavailability of a globally optimized query plan. Even in a closed database environment, to globally

optimize triggers would still be NP-complete, as shown in [95]. However, CQ systems may apply some of the principles to locally optimize triggers and queries for more efficient execution of continual queries.

Another set of techniques are among adaptive query processing [9, 71, 50], in which they try to provide efficient query execution by changing query plans according to incoming data changes.

For large data sets, it is often needed to get fast approximate answers. Online query answering mechanisms [46] are used to allow users to both observe the progress of their aggregation queries and control query execution on the fly. This kind of systems can be also called approximate query answering systems, which can return approximate query results continuously to the user instead of giving a precise result upon finishing. This is useful when exact results are not required by applications, while the user can still view a sketch of the query results and has the flexibility to control further query execution. [35] studied approximate computation techniques for correlated aggregates. Data reduction [12] is closely associated with aggregation, with the recent emergence of data cube research and online analytical processing.

To make use of previous query results when evaluating similar queries or re-evaluating the same query, query and predicate caching techniques [3, 58, 53] are commonly used for faster evaluation of queries.

Since a CQ system uses mediator/wrapper architecture in an open environment like the World Wide Web, Web proxy caching techniques [17, 122] can be used to shorten response time when queries involve fetching pages from remote sites.

Another suite of useful techniques include efficient information dissemination [6, 119] and notification mechanisms [82].

Given the runtime characteristics of continual query systems, it is important for us to study grouping and optimization opportunities at multiple levels since the processing of continual queries is multi-stage (event detection, trigger evaluation, query execution, and notification). Although some of the previous studies addressed grouping for triggers (e.g., TriggerMan [43]) or continuous queries with triggers (e.g., NiagaraCQ [25, 24]), to the

best of our knowledge, they have yet to exploit the opportunities for multi-level processing optimization. One of the main contributions of this paper is showing that multi-level optimization performs better than using a single grouping mechanism.

We have also identified that the benefit of grouping depends critically (by two orders of magnitude) on the group size distribution. Thus, the granularity of grouping (number of groups resulting from grouping) is important for the processing of continual queries, which needs flexible algorithms that can adjust group sizes according to different system loads. Accordingly, we propose trigger grouping algorithms with different grouping granularity, namely grouping for S-type, C-type, and OC-type triggers.

Data stream systems have emerged recently due to the increasing need in many applications such as stock trading and traffic monitoring. Different stream system models, architectures, and challenges are studied in [10, 123, 11, 12, 20, 35, 70, 71, 81, 98, 121, 123]. Data stream processing is a natural application of *continual query* systems.

6.3 Web Page Monitoring

There has been considerable research done on data update monitoring in databases. Powerful database techniques such as active databases and materialized views have been studied extensively. These techniques have been proposed primarily for "data-centric" environments, where data is well organized and centrally controlled in a database with a close-world assumption. When applied to an open information universe as the Internet, these assumptions no longer hold, and some of the techniques do not easily extend to scale up to the distributed interoperable environment.

Comparing with the state-of-art of research in active databases, the WebCQ system differs in three ways: First, the WebCQ system targets at monitoring and tracking changes to arbitrary web pages. Second, WebCQ monitors data provided by the content providers on remote servers, and WebCQ monitoring and tracking service requires neither the control over the data it monitors nor the structural information about the data it is tracking. Whereas active databases can only monitor structured data that reside in a database. Third, the WebCQ system provides efficient and scalable proxy service as well as grouping techniques

for trigger processing, and it can handle large numbers of concurrently running sentinels on a large number of web pages.

There have been several systems developed for monitoring source data changes in the past. One type of systems is the extension of Web search engines or search software by monitoring URL changes and notifying the users whenever the URLs of the data sources of interest have changed. A representative system is the NetMind (`http://www.netmind.com/`, formerly known as URL-minder), which provides keyword-based and phase-based change detection and notification service over Web pages. Compared with WebCQ system, NetMind does not provide fine grain of specification on triggering conditions. The content-based condition is still keyword-based. They do not support up-to-minute update monitoring either. The second type of monitoring systems is the application-specific change notification systems such as E*Trade alert facility, Amazon.com new book notification service, and so on. The most representative one of this type is the Stanford news service SIFT [120], which filters and notifies Internet news of interest through user preferences on news items and news groups. The problem with this type of systems is that they are tailored to a particular (type of) data source or application, which consequently do not have the generality and extensibility as WebCQ. The third type of projects is the change detection over wrapped Web pages, such as the *C*3 [23] project at Stanford and OpenCQ [62]. C3 develops a query subscription mechanism that allows users to subscribe the data sources they are interested in detection of changes as well as query over the change databases. The C3 system periodically goes to the subscribed web sites, fetches the pages, applies the HTML-based `diff` functions to derive the types of changes, and archives the changes in the change databases. The OpenCQ system is specialized in tracking changes over structured information. Therefore, wrappers [61, 67] are needed to transform Web pages into structured data format and then install continual queries over the structured data. In contrast, WebCQ is designed to monitor and tracking arbitrary web pages directly. No structured formats are required. The advantage is obvious with respect to the system scalability and effectiveness. A disadvantage compared with OpenCQ is that it does not support semantically-enhanced and fine-grained information monitoring, such as notify me when both IBM stock price and

Microsoft stock price drop by 10% within one week.

In recent years, increasing interest in online page change monitoring services has been shown. Despite the common interests, there are few available systems providing automatic and integrated page change detection and notification. Some currently working systems include ChangeDetection [21], InfoMinder [49], TrackEngine [105], WatchThatPage [106], and WebCQ [109]. Some past and notable systems include NetMind (out of service and bought by Pumatech) and AT&T's AIDE [30]. There are also client side tools available to organize Web bookmarks and monitor Web site changes (e.g. [110, 111]). However, compared to server side approaches, these tools are not scalable in the context of Internet-Scale change monitoring. The above-mentioned page monitoring systems differ in quality of difference generation and summarization as well as methods of notification delivery (for example TrackEngine sends the entire change content to users through emails while most others send only summaries of changes). Most of them have limited capabilities in expressiveness, usability, detection accuracy, and scalability. Furthermore, it is unclear how these systems integrate their services into business processes. There is limited information available publicly for the authors to conduct further investigation on techniques employed by these systems. Compared to other server side approaches, WebCQ have advantages in the following aspects:

- Richer change semantics through various sentinel types
- More accurate difference generation and summarization
- Response actions in WebCQ (current HTTP implementation)
- Enhanced difference presentation and visualization
- Easy integration between monitoring and business processes for better business information exchange experience

We are working on expanding the service to cover more types of information sources (semi-structured textual sources including PDF, Doc, XML, etc.) by utilizing more advanced object extraction techniques [61] and conversion tools. Our future work includes

extending the framework for Web site and page collection monitoring (similar to [16]). Data mining and machine learning approaches will be incorporated into the framework. We are also exploring and developing tools for better change visualization. Existing research can be applied to the value-added services [26, 27, 36, 56, 77].

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

This dissertation documents the design and implementation of the first open-source large-scale information monitoring systems, namely the *OpenCQ* system for structured and semi-structured information monitoring, and the *WebCQ* system for arbitrary Web page monitoring. The thesis research is experimental computer systems research. The dissertation archives the complete set of technical solutions we have developed for building an Internet-Scale information monitoring system. These solutions include:

- Using *Continual Queries* as a means to model information monitoring requests;
- A multi-level optimization technique in CQ processing, namely trigger grouping, query result caching, and notification grouping;
- The *Differential Re-evaluation Algorithm* to optimize CQ execution and save re-computation of continual query evaluations;
- A set of tools and facilities as building blocks for constructing a large-scale information monitoring system

In this dissertation, we first present the system level facilities for building an Internet-scale continual query system, including the design and development of two operational CQ monitoring systems OpenCQ and WebCQ, the engineering issues involved, and our solutions. We then describe a number of research challenges that are specific to large-scale information monitoring and the techniques developed in the context of OpenCQ and WebCQ to address these challenges. Example issues include how to efficiently process large number of continual queries, what mechanisms are effective for building a scalable distributed trigger system that is capable of handling tens of thousands of triggers firing at hundreds of data sources, how to effectively disseminate fresh information to the right users

at the right time. We have developed a suite of techniques to optimize the processing of continual queries, including an effective CQ grouping scheme, an auxiliary data structure to support group-based indexing of CQs, and a differential CQ evaluation algorithm (DRA) for incrementally computing new query results from updates on top of previous results. The third contribution is the design of an experimental evaluation model and testbed to validate the solutions. We have engaged our evaluation using both measurements on real systems (OpenCQ/WebCQ) and simulation-based approach. We also discuss the issues with how to apply information monitoring systems for automating change response in business information exchanges.

To our knowledge, the research documented in this dissertation is to date the first one to present a focused study of research and engineering issues in building large-scale information monitoring systems using continual queries.

7.1 Future Work

The research presented in this thesis lays out the foundations for studying large-scale information monitoring systems. Two prototype systems (OpenCQ and WebCQ) have been built to validate the proposed solutions. However, the work has just began. There are many issues that I would like to continue working on.

Some of the ongoing work include experiments to evaluate the performance of alternative architectures and algorithms and the performance improvements by incorporating advanced indexing techniques into the CQ systems, and utilizing research results in incremental query evaluation and multiple query optimization in the framework. I am also investigating the scalability enhancement of the *Continual Query* systems along three dimensions: replication, distribution, and caching.

The most recent focus is on extending WebCQ system to address increasing users' needs for Web page monitoring, including more accurate difference generation and presentation algorithms, richer result visualization, result summarization, presentation, and content adaptation for mobile (PDA) users, and collection-based Web page/site monitoring.

Further development to ensure accurate results and to assist users in formulating appropriate responses to the changes is also needed to continue to increase the value of the monitoring system. Along this line of work are change measurement, change relevance assessment, and automated change response services. These techniques are essential to orchestrate a full-fledged *change management system* with *change monitoring systems* such as OpenCQ and WebCQ.

APPENDIX A

CONTINUAL QUERY SPECIFICATION

This is the original *Continual Query* specification used in OpenCQ prototype implementation.

```
<ContinualQuerySpec> ::=
            CREATE CQ [<ContinualQueryName>] AS
                Query: <Query>
                Trigger: <TrigCond>
                Start: <StartCond>
                Stop:  <StopCond>
                NotifyCondition: <NotifyCond>
                [NotifyMethod: <methodType><methodSignature>]
                [Calendar: <calendaExpression>]

<ContinualQueryName> := String
<Query> ::= EXTRACT <SelectList>
            FROM <ObjectList>
            [CONDITION <SearchCondition>]
            [GROUP BY <AttributeList>]
            [ORDER BY <SortSpecList>]
<TriggerCond> ::= <TimeTriCond> | <ContentTriCond>
<StartCond> ::= <TimePoint>
<StopCond> ::=  <TimePoint>
<TimePoint> ::= <Month> '-' <Day> '-' <Year> ' ' <Hour> ':' <Min> ' ' <TimeZone>
<SelectList> ::= * | <AttributeList>
<AttributeList>:= <Attribute> | <Attribute> [, <AttributeList>]
<Attribute> ::= id | <ObjectName>.<AttributeName> | <AggreSpec>
<ObjectName> ::= String
<AggreSpec> ::= COUNT(*) | <AggreFunc>(<Attribute>)
<AggreFunc> ::= AVG | MAX | MIN | SUM | COUNT
<ObjectList> ::= <ObjectName> | <ObjectName> [, <ObjectList>]
<SearchCondition> ::= <BoolExpr> | [NOT] <BoolExpr>
<BoolExpr> ::= <BoolTerm> | <BoolTerm> <LogicOp> <BoolExpr>
<LogicOp> ::= AND | OR
<BoolTerm> ::= <Predicate> | <ValueExpr>
```

```
<Predicate> ::= <ValueExpr> <Op> <ValueExpr>
<Op> ::= <CompOp> | [NOT] <LikeOp>
<CompOp> ::= <> | = | < | > | <= | >=
<LikeOp> ::= LIKE | CONTAINS
<ValueExpr> ::= <NumValExpr> | <StrValExpr>
<NumValExpr> ::= <Term> | <NumValExpr> + <Term> | <NumValExpr> - <Term>
<Term> ::= <Factor> | <Term> * <Factor> | <Term> / <Factor>
<Factor> ::= [+|-] | <NumValue>
<NumValue> ::= number | <Attribute> | (<NumValExpr>)
<StrValExpr> ::= <StrValue>
<StrValue> ::= String | <Attribute>
<SortSpecList> ::= <SortSpec> | <SortSpec> [, <SortSpec> ]
<SortSpec> ::= <Attribute> [<OrderKey>]
<OrderKey> ::= ASC | DESC
<TimeTriCond> ::= <MinExpr> '&&' <HourExpr> '&&' <DayOfMonExpr> '&&'
                  <MonthExpr> '&&' <DayOfWeekExpr>
<MinExpr> ::= <MinFactor> | <MinFactor> [, <MinExpr>] | <NotSpecified>
<MinFactor> ::= <MinVal> | <MinVal> - <MinVal>
<NotSpecified> ::= null
<MinVal> ::= 00..59
<HourExpr> ::= <HourFactor> | <HourFactor> [, <HourExpr>] | <NotSpecified>
<HourFactor> ::= <HourVal> | <HourVal> - <HourVal>
<HourVal> ::= 00..23
<DayOfMonExpr> ::= <DayOfMonFactor> | <DayOfMonFactor> [, <DayOfMonExpr>]
                   | <NotSpecified>
<DayOfMonFactor> ::= <DayOfMonVal> | <DayOfMonVal> - <DayOfMonVal>
<DayOfMonVal> ::= 1..31
<MonthExpr> ::= <MonthFactor> | <MonthFactor> [, <MonthExpr>] | <NotSpecified>
<MonthFactor> ::= <MonthVal> | <MonthVal> - <MonthVal>
<MonthVal> ::= 1..12
<DayOfWeekExpr> ::= <DayOfWeekFactor> | <DayOfWeekFactor> [, <DayOfWeekExpr>]
                    | <NotSpecified>
<DayOfWeekFactor> ::= <DayOfWeekVal> | <DayOfWeekVal> - <DayOfWeekVal>
<DayOfWeekVal> ::= 0..6
<ContentTriCond> ::= <ContTriGroup> | <ContTriGroup> <EventOp> <ContentTriCond>
<ContTriGroup>  ::= <AtClause> <ContPrimitive> <GrpConstraint>
<AtClause> ::= AT SOURCE <URL>
<ContPrimitive> ::= WHEN [<AggreFunc>(]<ObjectName>.<Attribute>[)] <ContTriCondOp>
                    [ <Value> ]
<GrpConstraint> ::= WHERE <ContPrimitiveList> [GROUPBY <AttributeList>]
<ContPrimitiveList> ::= <ContPrimitive> | <ContPrimitiveList>
```

```
                        <GrpJointOp> <ContPrimitive>
<EventOp> ::= AND | OR | <sequence> | <parallel>
<ContTriCondOp> ::= <> | = | < | > | <= | >= | CHANGES | CONTAINS | LIKE
                    | INCBY | DECBY | INCBYP | DECBYP
<GrpJointOp> ::= ^ | v
<ContTriVal> ::= String
<Year> ::= [1-9][0-9][0-9][0-9]
<Month> ::= <MonthVal>
<Day> ::= <DayOfMonVal>
<Hour> ::= <HourVal>
<Min> ::= <MinVal>
<TimeZone> ::= [-|+][0-1][0-9]
```

APPENDIX B

WEBCQ SENTINEL SPECIFICATION

This is the specialized *Continual Query* specification in WebCQ prototype implementation. In WebCQ, a continual query is called a *WebCQ Sentinel*. In this specification, texts after '//' are considered comments.

```
<WebCQ Sentinel> ::=
     CREATE Sentinel [<sentinel name>] AS
         Query: <query>
         Trigger Condition: <time interval>
         Start Condition: <time point>
         Stop Condition: <time point>
         Notification Condition: <time interval>
         [Notification Method: <method type><method signature>]

<sentinel name>::= String
<query> ::= {Sentinel Type: <sentinel type>
             Sentinel Target: <sentinel target>
             Sentinel Object: <object desc>}
<sentinel type>::= <HTML-specific-type> | <general-type> | <rule-type>
<HTML-specific-type> ::= 'All Links' | 'All Images' | 'Table' | 'List'
<general-type> ::= 'Any Change' | 'Phrase' | 'All Words' | 'Keyword'
<rule-type> ::= <regular-expression>
<sentinel target>::= <URL>
<object desc>::= String
<time interval>::= Integer {'MINUTE' | 'HOUR' | 'DAY' | 'WEEK'}
<time point>::= <Month> '-' <Day> '-' <Year> ' ' <Hour> ':' <Min> ' ' <TimeZone>
<Year> ::= 20{03..99}
<Month> ::= 1..12
<Day> ::= 1..31
<Hour> ::= 00..23
<Min> ::= 00..59
<TimeZone> ::= [-|+][0-1][0-9]
<method type>::= 'EMAIL'
                 | 'WEB BOARD'
```

```
                | 'LOGFILE'
                | 'EXTERNAL'
                | 'FAX'
                | 'PAGER'
// Regular Expression BNF
// Adapted from http://www.faqts.com/knowledge_base/view.phtml/aid/25718/fid/200
<regular-expression> ::= <term> | <regular-expression>
<term> ::= <factor>
           | <factor> <term>
<factor> ::= <atom>
             | <atom> <meta character>
<atom> ::= '.'
           | '(' regular-expression ')'
           | '[' character-set ']'
           | '[' ^character-set ']'
           | '{' min '}'
           | '{' min ',' '}'
           | '{' min ',' max '}'
<character-set> ::= <character-item>
                      | <character-item> <character-set>
<character-item> ::= <character>
                      | <character> '-' <character>
<character> ::= <anycharacter-except-metacharacters>
                | '\'<anycharacter-except-specialcharacters>
<metacharacter> ::= '?'    // non-greedy match
                  | '*'    // 0 or more, greedy
                  | '*?'   // 0 or more, non-greedy
                  | '+'    // 1 or more, greedy
                  | '+?'   // 1 or more, non-greedy
                  | '^'    // begin of line character
                  | '$'    // end of line character
                  | '$`'   // the characters to the left of the match
                  | '$''   // the characters to the right of the match
                  | '$&'   // the characters that are matched
                  | '\t'   // tab character
                  | '\n'   // newline character
                  | '\r'   // carriage return character
                  | '\f'   // form feed character
                  | '\cX'  // control character CTRL-X
                  | '\N'   // the characters in Nth tag (if on match side)
                  | '$N'   // the characters in Nth tag (if not on match side)
```

```
                    | '\NNN' // octal code for character NNN
                    | '\b'   // match a 'word' boundary
                    | '\B'   // match not a 'word' boundary
                    | '\d'   // a digit, [0-9]
                    | '\D'   // not a digit, [^0-9]
                    | '\s'   // whitespace, [ \t\n\r\f]
                    | '\S'   // not a whitespace, [^ \t\n\r\f]
                    | '\w'   // 'word' character, [a-zA-Z0-9_]
                    | '\W'   // not a 'word' character, [^a-zA-Z0-9_]
                    | '\l'   // lowercase next char
                    | '\u'   // uppercase next char
                    | '\L'   // change characters to uppercase, until \E
                    | '\U'   // change characters to uppercase, until \E
                    | '\E'   // end case modification
                    | '\Q'   // put a quote (de-meta) on characters, until \E
<min> ::= Integer
<max> ::= Integer
<anycharacter> ::= ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / | : | ;
                   | < | = | > | ? | @ | [ | \ | ] | ^ | _ | ` | { | } | ~ |<verBar>
                   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
                   | A | B | C | D | E | F | G | H | I | J | K | L | M | N
                          | O | P | Q | R | S | T | U | V | W | X | Y | Z
                   | a | b | c | d | e | f | g | h | i | j | k | l | m | n
                          | o | p | q | r | s | t | u | v | w | x | y | z
<verBar> ::= '|'
```

APPENDIX C

WEBCQ RELATED CONCEPTS

The following explains some basic concepts used in WebCQ object extraction.

1. **Document**

   A *Document* is referring to the Web page itself.

2. **Link**

   A *Static Web page* is a Web page that does not incur server-side computing. A static Web page does not change between subsequent page fetching requests unless the owner of the page manually modifies it.

   A *Dynamic Web page* is a Web page that involves server-side computing for content generation. A server-side program is invoked to create the content of the page whenever a page request is received at the Web server. Example of dynamic Web pages include JSP (Java Server Page), ASP (Active Server Page), SSI (Server-Side Includes) pages, and pages generated by CGI (Common Gateway Interface) programs or Java servlets.

   A *Link* object can be classified into the following categories:

   - Any URL[1] specified in an HTML tag that has "href" parameter, but not a URL of "mailto://" or "javascript://" types.
   - Any HTML form action, i.e., the value specified for the "action" parameter inside <FORM> tag.
   - Any URL that is the content for "value" parameter within any HTML tag, e.g., "<OPTION VALUE=http://www.cc.gatech.edu>". Some Web pages use this

[1]Right now, we recognize these URL protocols: http, ftp, file, gopher, mailto, mid, afs, cid, news, nntp, prospero, telnet, rlogin, tn3270, and wais. See ftp://ftp.isi.edu/in-notes/rfc1738.txt for more details.

kind of links inside a drop-down selection menu.

3. **Image**

   An *Image* object is identified by the "<IMG>" HTML tag.

4. **Table**

   A *Table* data object is recognized by the "<TABLE>" and "< /TABLE>" HTML tags.

5. **List**

   A *List* data object is identified by "<UL>"-"< /UL>", "<OL>"-"< /OL>", or "<DL>"-"< /DL>" tags. *All Words* object represent all the "words" on the page.

6. **Word**

   An ASCII character string[2] starting with alphabetic character ('a'-'z', 'A'-'Z') or a numerical value ('0'-'9') without any whitespace (horizontal tab - %x09, space - %x20, line feed - %x0A, carriage return - %x0D). For example, "WebCQ" is considered a word, as well as "2000-7-6,".

7. **Phrase**

   Phrase data object is defined as a set of words concatenated by zero or more white spaces, which does not contain HTML tags.

8. **Keyword**

   Keyword data object is consisted of one or multiple words which does not contain HTML tags.

9. **Regexp text segment**

   A *Regexp text segment* is a segment of HTML document source represented using a regular expression, e.g., "Daily stock quotes(.*?)IBM(.*?)Volume". For each extracted

[2]Right now, we do not consider extended ASCII.

object (data object), we have an associated sentinel type for the detection of changes to the particular object.

APPENDIX D

WEBCQ SENTINEL CHANGE DETECTION ALGORITHM

```
Document D_new, D_cache;      // the current and cached Web pages
int ε;                        // a threshold number (> 0), default 1
Case Sentinel Type of:
        "Any change":
                if (timestamp(D_new) <> timestamp(D_cache)) then {
                        if (|sizeOf(D_new) - sizeOf(D_cache)| > ε)
                                then detected "document changed";
                                else if (MD5(D_cache) <> MD5(D_new) then detected "document changed";
                }
                return;
        "All links"
                if ({links in D_new} MINUS {links in D_cache} <> φ)
                        then detected "new links added";
                if ({links in D_cache} MINUS {links in D_new} <> φ)
                        then detected "old links removed";
                return;
        "All images"
                if ({images in D_new} MINUS {images in D_cache} <> φ)
                        then detected "new images added";
                if ({images in D_cache} MINUS {images in D_new} <> φ)
                        then detected "old images removed";
                return;
        "Phrase": // for simplicity, we do not consider duplications of phrase match
                String p = Phrase_sentinel_content; //attribute of sentinel object
                String B = $Beginning_part_of_bounding_box; //initialized from cache
                String E = $Ending_part_of_bounding_box; //initialized from cache
                if (B ∈ D_new and E ∈ D_new)
                        then {// bounding boxes are not changed
                                //extract(D,B,E,ifTag) is a function to get the text region inside a document D given
                                //bounding box bounds B and E, boolean "ifTag" controls whether to include tags or not
                                String O_p = extract(D_new, B, E, true);
                                if (O_p <> O_p_cache) then detected "phrase change type (1) ";
                        } else { // bounding boxes are changed
                                //expand(D,p) is a function to get the text region in a document D if its "non-tag"
                                // representation is p
                                String O'_p = expand(D_new, p);
                                if (E ∈̄ D_new && B ∈̄ D_new) { //both B and E changed
                                        if (O'_p <> O_p_cache) then
                                                B' = adjust(B); E' = adjust(E);
                                                if (extract(O'_p,B',E',false) <> extract(O_p_cache,B',E',false))
                                                        then detected "phrase change type (7)";
                                } else if (E ∈̄ D_new) then { // only E changed
                                        if (O'_p <> O_p_cache) then
                                                E' = adjust(E);
                                                if (extract(O'_p,φ,φ,false) <> extract(O_p_cache,φ,φ,false))
                                                        then detected "phrase change type (6)";
                                } else if (B ∈̄ D_new) { // only B changed
                                        if (O'_p <> O_p_cache) then
                                                B' = adjust(B);
                                                if (extract(O'_p,φ,φ,false) <> extract(O_p_cache,φ,φ,false))
                                                        then detected "phrase change type (5)";
                                }
                        }
                return;
```

Figure 41: WebCQ Sentinel Change Detection Algorithm

```
"Table":
        String t = Table_sentinel_content; //attribute of sentinel object
        String B = $Beginning_part_of_bounding_box; //initialized from cache
        String E = $Ending_part_of_bounding_box; //initialized from cache
        String O_t = extract(D_new, B, E, true);
        if (O_t <> null and toString(table_cache) <> O_t))
                then detected "table change";
                //* we can further detect detailed table cell changes
                else if (O_t == null)
                        then
                        //* it could be bounding box change or table disappearance, we need to distinguish
        return;
"List":
        String l = List_sentinel_content; //attribute of sentinel object
        String B = $Beginning_part_of_bounding_box; //initialized from cache
        String E = $Ending_part_of_bounding_box; //initialized from cache
        String O_l = extract(D_new, B, E, true);
        if (O_l <> null and toString(list_cache) <> O_l))
                then detected "list change";
                // we can further detect detailed list item changes
                else if (O_l == null)
                        then
                        //* it could be bounding box change or list disappearance, we need to distinguish
        return;
"All words"
        if ({words in D_new} MINUS {words in D_cache} <> φ)
                then detected "new words added";
        if ({words in D_cache} MINUS {words in D_new} <> φ)
                then detected "old words removed";
        return;
"Key words": // case-insensitive and boolean "AND" match
        κ = {keyword list};
        flag = F_cache; // previous keyword match result
        match = false;
        foreach kw ∈ κ do {
                if (∃ w ∈ {All words} s.t. UPPERCASE(kw) is_substring_of UPPERCASE(w))
                        then match = true;
                        else { match = false; break; }
        if (flag <> match) then {
                if (flag == false)
                        then detected "keyword appeared";
                        else detected "keyword disappeared";
        }
        F_cache = match;
        return;
"Regular Expression":
        String regtext = eval($regexp);
        if (regtext <> regtext_cache) then
                detected "content update to regexp segment";
        return;
```

Figure 42: WebCQ Sentinel Change Detection Algorithm (continued)

APPENDIX E

DRA - LEMMAS AND PROPOSITION PROOFS

Lemma 1 The following equations hold:

1. $\texttt{attach}[\texttt{insertions}(\Delta R), \texttt{new}(\mathcal{R})] = \texttt{insertLog}(\Delta R)$.
2. $\texttt{attach}[\texttt{deletions}(\Delta R), \texttt{old}(\mathcal{R})] = \texttt{deleteLog}(\Delta R)$.
3. $\texttt{Outerjoin}_{\texttt{tid}}(\texttt{deleteLog}(\Delta R), \texttt{insertLog}(\Delta R)) = \Delta R$.
4. $R - \texttt{combine}(R, \Delta R) = \texttt{deletions}(\Delta R)$.
5. $\texttt{combine}(R, \Delta R) - R = \texttt{insertions}(\Delta R)$. □

The proof of the equations in Lemma 1 follows straightforward from Definition 2 and Definition 3.

Lemma 2 Let the differential substitution list SL be $\{[R_1, \Delta R_1],\ldots,[R_n, \Delta R_n]\}$ $(n \geq 1)$. The following equations hold:

1. $\texttt{Propagate}(R;\ [R, \Delta R]) = \Delta R$.
2. $\texttt{combine}(\mathrm{Q}(R_1, \ldots, R_n), \texttt{Propagate}(\mathrm{Q};\ \mathrm{SL})) = \mathrm{Q}(R'_1, \ldots, R'_n)$. □

Proposition 1 $\texttt{Propagate}(\sigma[R;F];(R, \Delta R)) \equiv \texttt{DiffSelect}[\Delta R;F]$.

Proof

Let RHS and LHS represent the right and left hand side of the equation. Let the condition `tid` of the $\texttt{Outerjoin}_{\texttt{tid}}$ denote "$\texttt{tid}^{<old>} = \texttt{tid}^{<new>}$", $F^{<old>}$ and $F^{<new>}$ denote the same predicate with all attribute names superscripted accordingly. By Definition 6,

RHS = $\texttt{Outerjoin}_{tid}(\sigma[\texttt{deleteLog}(\Delta R);F^{<old>}],\ \sigma[\texttt{insertLog}(\Delta R);F^{<new>}])$.

Let E1 = $\sigma[R;F] - \sigma[\texttt{combine}(R, \Delta R);F] = \sigma[R - \texttt{combine}(R, \Delta R);\ F]$

$= \sigma[\texttt{deletions}(\Delta R);\ F]$ (by Lemma 1(4)).

E2 = $\sigma[\texttt{combine}(R, \Delta R);F] - \sigma[R;F] = \sigma[\texttt{combine}(R, \Delta R) - R;\ F]$

$= \sigma[\texttt{insertions}(\Delta R);\ F]$ (by Lemma 1(5)).

By Definition 5 (`Propagate`) and Definition 4 (`Diff`),

LHS = $\texttt{Diff}(\sigma[R;F];\sigma[\texttt{combine}(R,\Delta R);F])$

$= \texttt{Outerjoin}_{tid}(\texttt{attach}[\text{E1};\texttt{old}(\mathcal{R})], \texttt{attach}[\text{E2};\texttt{new}(\mathcal{R})])$

$= \texttt{Outerjoin}_{tid}(\sigma[\texttt{deleteLog}(\Delta R);F^{<old>}], \sigma[\texttt{insertLog}(\Delta R);F^{<new>}])$ (Definition 3). □

Proposition 2 $\texttt{Propagate}(\Pi[R;\mathbf{X}]; (R,\Delta R)) \equiv \texttt{DiffProj}[\Delta R; \texttt{old}(\mathbf{X}) \bigcup \texttt{new}(\mathbf{X})]$.

Proof

Let E1 = $\Pi[R;\mathbf{X}] - \Pi[\texttt{combine}(R,\Delta R);\mathbf{X}] = \Pi[R - \texttt{combine}(R,\Delta R);\mathbf{X}]$

$= \Pi[\texttt{deletions}(\Delta R);\mathbf{X}]$ (Lemma 1(4)).

E2 = $\Pi[\texttt{combine}(R,\Delta R);\mathbf{X}] - \Pi[R;\mathbf{X}] = \Pi[\texttt{combine}(R,\Delta R) - R;\mathbf{X}]$

$= \Pi[\texttt{insertions}(\Delta R);\mathbf{X}]$ (Lemma 1(5)).

The expression $\Pi[R \bigcap \texttt{combine}(R,\Delta R);\text{X}]$ returns the set of $\mathbf{X}$ tuples for which their attributes are unchanged. By Definition 7,

LHS = $\texttt{Diff}(\Pi[R;\mathbf{X}], \Pi[\texttt{combine}(R,\Delta R);\mathbf{X}])$

$= \texttt{Outerjoin}_{tid}(\texttt{attach}[\text{E1};\texttt{old}(\mathbf{X})], \texttt{attach}[\text{E2};\texttt{new}(\mathbf{X})])$

$= \texttt{Outerjoin}_{tid}(\Pi[\texttt{deleteLog}(\Delta R);\texttt{old}(\mathbf{X})], \Pi[\texttt{insertLog}(\Delta R);\texttt{new}(\mathbf{X})])$

(by Lemma 1(1)(2) and Definition 3)

$= \sigma_P(\Pi[\Delta R;\texttt{old}(\mathbf{X}) \bigcup \texttt{new}(\mathbf{X})])$ (by Lemma 1(3) and Definition 7) = RHL. □

In the proof we have used the distributive property of projection over difference (i.e., $\Pi_X(R_1 - R_2) = \Pi_X(R_1) - \Pi_X(R_2)$), which holds under the assumption that each tuple has a unique immutable identifier (recall Section 4.2.1). This is because, under this assumption, the projection is re-defined as $\Pi_X(R) = \{t' | t'.tid = t.tid \wedge t'[X] = t[X]\}$, and the result relation scheme is $\{tid\} \cup X$, instead of X.

However, the distributive property of projection over difference does not hold for the traditional relational project operator which eliminates the duplicates in its result relation [72]. That is $\Pi_X(R_1 - R_2) \neq \Pi_X(R_1) - \Pi_X(R_2)$. An alternative way to maintain the distributive property of projection over difference is to associate a multiplicity *counter* to each tuple in a relation. This approach has been used by many algorithms for updating

materialized views [14, 44].

Proposition 3 Let the differential substitution list SL be $\{[R_1, \Delta R_1],[R_2, \Delta R_2]\}$.

$$\texttt{Propagate}(\bowtie_{P_{join}}(R_1, R_2);\ [(R_1, \Delta R_1),(R_2, \Delta R_2)]) \equiv \texttt{DiffJoin}_{P(tid_1,tid_2)}(R_1, R_2).$$

Proof

$$\begin{aligned}
\text{Let E1} &= \bowtie_{P_{join}}(R_1, R_2) - \bowtie_{P_{join}}(\texttt{combine}[R_1, \Delta R_1],\texttt{combine}[R_2, \Delta R_2])) \\
&= \bowtie_{P_{join}}(\texttt{deletions}(\Delta R_1),R_2) \bigcup \bowtie_{P_{join}}(R_1,\texttt{deletions}(\Delta R_2)) \\
&\qquad \bigcup \bowtie_{P_{join}}(\texttt{deletions}(\Delta R_1),\texttt{deletions}(\Delta R_2)). \\
\text{E2} &= \bowtie_{P_{join}}(\texttt{combine}[R_1, \Delta R_1],\texttt{combine}[R_2, \Delta R_2])) - \bowtie_{P_{join}}(R_1, R_2) \\
&= \bowtie_{P_{join}}(\texttt{insertions}(\Delta R_1),R_2) \bigcup \bowtie_{P_{join}}(R_1,\texttt{insertions}(\Delta R_2)) \\
&\qquad \bigcup \bowtie_{P_{join}}(\texttt{insertions}(\Delta R_1),\texttt{insertions}(\Delta R_2)).
\end{aligned}$$

By Definition 5 and Definition 4,

$$\begin{aligned}
\text{LHS} &= \texttt{Diff}(\bowtie_{P_{join}}(R_1, R_2),\ \bowtie_{P_{join}}(\texttt{combine}[R_1, \Delta R_1],\texttt{combine}[R_2, \Delta R_2])) \\
&= \texttt{Outerjoin}_{P(tid_1,tid_2)}(\texttt{attach}[\text{E1};\ \texttt{old}(\mathcal{R})\bigcup\texttt{new}(\mathcal{R})],\ \texttt{attach}[\text{E2};\ \texttt{old}(\mathcal{R})\bigcup\texttt{new}(\mathcal{R})]) \\
&= \texttt{Outerjoin}_{P(tid_1,tid_2)}(\texttt{attach}[\bowtie_{P_{join}^{<old>}}(\texttt{deletions}(\Delta R_1),\ R_2);\ \texttt{old}(\mathcal{R})], \\
&\qquad\qquad \texttt{attach}[\bowtie_{P_{join}^{<new>}}(\texttt{insertions}(\Delta R_1),\ R_2);\ \texttt{new}(\mathcal{R})]) \\
&\quad \bigcup \texttt{Outerjoin}_{P(tid_1,tid_2)}(\texttt{attach}[\bowtie_{P_{join}^{<old>}}(R_1,\texttt{deletions}(\Delta R_2));\ \texttt{old}(\mathcal{R})], \\
&\qquad\qquad \texttt{attach}[\bowtie_{P_{join}^{<new>}}(R_1,\texttt{insertions}(\Delta R_2));\ \texttt{new}(\mathcal{R})]) \\
&\quad \bigcup \texttt{Outerjoin}_{P(tid_1,tid_2)}(\bowtie_{P_{join}^{<old>}}(\texttt{deleteLog}(\Delta R_1),\texttt{deleteLog}(\Delta R_2)), \\
&\qquad\qquad \bowtie_{P_{join}^{<new>}}(\texttt{insertLog}(\Delta R_1),\texttt{insertLog}(\Delta R_2)) \ = \ \text{RHS}.
\end{aligned}$$

□

REFERENCES

[1] Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. L., "The Lorel query language for semistructured data," *International Journal on Digital Libraries*, vol. 1, no. 1, pp. 68–88, 1997.

[2] Acharya, S., Franklin, M., and Zdonik, S., "Balancing push and pull for data broadcast," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (Tucson, Arizona), May 1997.

[3] Adali, S., Candan, K., Papakonstantinou, Y., and Subrahmanian, V., "Query caching and optimization in distributed mediator system," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, vol. 25, (Montreal, Quebec, Canada), 1996.

[4] Aguilera, M., Strom, R., Sturman, D., Astley, M., , and Chandra, T., "Matching events in a content-based subscription system," in *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC)*, 1999.

[5] Alonso, R., Barbara, D., and Garcia-Molina, H., "Data caching issues in an information retrieval system," *ACM Transactions on Database Systems*, vol. 15, pp. 359–384, September 1990.

[6] Altinel, M. and Franklin, M. J., "Efficient filtering of xml documents for selective dissemination of information," in *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, (Cairo, Egypt), September 2000.

[7] ao Pereira, J., Fabret, F., Llirbat, F., and Shasha, D., "Efficient matching for web-based publish/subscribe systems," in *Proceedings of the International Conference on Cooperative Information Systems (CoopIS)*, 2000.

[8] Atzeni, P., Mecca, G., and Merialdo, P., "To weave the web," in *Proceedings of the 23rd International Conference on Very Large Databases (VLDB)*, (Athens, Greece), August 1997.

[9] Avnur, R. and Hellerstein, J., "Eddies: continuously adaptive query processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (Dallas, Texas), May 2000.

[10] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J., "Models and issues in data stream systems," in *Proceedings of ACM Symp. on Principles of Database Systems (PODS 2002)*, (Madison, Wisconsin), June 2002.

[11] Babu, S. and Widom, J., "Continuous queries over data streams," *ACM SIGMOD Record*, September 2001.

[12] Barbara, D., DuMouchel, W., Faloutsos, C., Haas, P. J., Hellerstein, J. M., Ioannidis, Y. E., Jagadish, H. V., Johnson, T., Ng, R. T., Poosala,

V., Ross, K. A., and Sevcik, K. C., "The new jersey data reduction report," *IEEE Data Engineering Bulletin*, vol. 20, no. 4, pp. 3–45, 1997.

[13] Barbara, D. and Alonso, R., "Processing continuous queries in general environments," tech. rep., Matsushita Information Technology Laboratory, Princeton, NJ, June 1993.

[14] Blakeley, J., Larson, P., and Tompa, F., "Efficiently updating materialized views," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, (Washington, DC), pp. 61–71, May 1986.

[15] BotSpot. http://bots.internet.com.

[16] Boyapati, V., Chevrier, K., Finkel, A., Glance, N., Pierce, T., and Whitmer, C., "Changedetector: A site-level monitoring tool for the www," in *Proceedings of WWW11*, (Honolulu, Hawaii), ACM, May 2002.

[17] Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S., "Web caching and zipf-like distributions: evidence and implications," *Proceedings of IEEE Infocom '99*, pp. 126–134, March 1999.

[18] Buneman, P., "Semistructured data," *ACM Symposium on Principles of Database Systems*, pp. 117–121, June 1997.

[19] Chakravarthy, S., "Architectures and monitoring techniques for active databases: An evaluation," in *Technical Report TR-92-041, University of Florida*, (Gainesville, FL), 1992.

[20] Chandrasekaran, S. and Franklin, M., "Streaming queries over streaming data," in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, (Hong Kong, China), 2002.

[21] ChangeDetection. http://www.changedetection.com.

[22] Chankunthold, A., P.B.Danzig, C.Neerdaels, and andK.J.Worrel, M., "A Hierarchical Internet Object Cache," *Proceedings of Usenix*, 1996.

[23] Chawathe, S., Abiteboul, S., and Widom, J., "Managing and querying changes in semi-structured data," in *Proceedings of ACM SIGMOD Conference*, 1997.

[24] Chen, J., DeWitt, D., and Naughton., J., "Design and evaluation of alternative selection placement strategies in optimizing continuous queries," in *Proceedings of the 18th International Conference on Data Engineering*, 2002.

[25] Chen, J., DeWitt, D., Tian, F., and Wang, Y., "Niagaracq: a scalable continuous query system for internet databases," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, (Dallas, Texas), May 2000.

[26] Chi, E. H., Pitkow, J., Mackinlay, J., Pirolli, P., Gossweiler, R., and Card, S. K., "Visualizing the evolution of web ecologies," in *Proceedings of the ACM SIGCHI Conference on human factors in computing systems*, (Los Angeles, CA), ACM, 1998.

[27] Crouch, D. B., Crouch, C. J., and Andreas, G., "The use of cluster hierarchies in hypertext information retrieval," in *Proceedings of Hypertext'89*, (Pittsburgh, Pennsylvania), November 1989.

[28] Dingle, A. and Partl, T., "Web cache coherence," *Fifth International World Wide Web Conference*, 1996.

[29] Douglis, F., Ball, T., Y.Chen, and Koutsofios, E., "WebGuide: Querying and Navigating Changes in Web Repositories," *Proceedings of 1996 USENIX Technical Conference*, pp. 1335–1344, January 1996.

[30] Douglis, F., Ball, T., Y.Chen, and Koutsofios, E., "The at&t internet difference engine: Tracking and viewing changes on the web," *World Wide Web Journal*, vol. 1, no. 1, pp. 27–44, 1998.

[31] Fabret, F., Jacobsen, H., Llirbat, F., Pereira, J., Ross, K. A., and Shasha, D., "Filtering algorithm and implementation for very fast publish/subscribe systems," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, (Santa Barbara, CA), May 2001.

[32] Fan, L., Cao, P., Almeida, J., and Broder, A. Z., "Summary cache: A scalable wide area web cache sharing protocol," *Proceedings of SIGCOMM*, 1998.

[33] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., and Berners-Lee, T., "RFC 2068: hypertext transfer protocol - HTTP/1.1," January 1997.

[34] Forgy, C., "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, vol. 19, pp. 17–37, September 1982.

[35] Gehrke, J., Korn, F., and Srivastava, D., "On computing correlated aggregates over continual data streams," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, (Santa Barbara, CA), May 2001.

[36] Glover, E. J., Tsioutsiouliklis, K., Lawrence, S., Pennock, D. M., and Flake, G. W., "Using web structure for classifying and describing web pages," in *Proceedings of WWW11 Conference*, (Honolulu, Hawaii), ACM, May 2002.

[37] GnuDiff. http://www.gnu.org/software/diffutils/.

[38] Gupta, A., Mumick, I., and Subrahmanian, V., "Maintaining views incrementally," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, (Washington, DC), pp. 157–166, May 1993.

[39] Gwetzman, J. and Seltzer, M., "The case for geographical pushing-caching," *HotOS Conference*, 1994.

[40] Haas, L., Chang, W., Lohman, G., McPherson, J., P.Wilms, Lapis, G., Lindsay, B., Pirahesh, H., Carey, M., and Shekita, E., "Starburst mid-flight: As the dust clears," *IEEE Transactions on Knowledge and Data Engineering*, pp. 377–388, March 1990.

[41] Hanson, E., "Rule condition testing and action execution in ariel," in *Proceedings of ACM SIGMOD Conference*, 1992.

[42] Hanson, E., Bodagala, S., and Chadaga, U., "Optimized trigger condition testing in ariel using gator networks," Technical Report 97-021, University of Florida CISE Dept., 1997.

[43] Hanson, E., Carnes, C., Huang, L., Konyala, M., and Noronha, L., "Scalable trigger processing," in *Proceedings of the International Conference on Data Engineering*, 1999.

[44] Hanson, E. N., "A performance analysis of view materialization strategies," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, (San Francisco, CA), pp. 440–453, May 1987.

[45] Hellerstein, J., "Optimization Techniques for Queries with Expensive Methods," *ACM Transactions on Database Systems*, no. To appear, 1998. Available at www.cs.berkeley.edu/ jmh.

[46] Hellerstein, J. M., Haas, P. J., and Wang, H. J., "Online aggregation," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (Tucson, Arizona), May 1997.

[47] Hypertext Transfer Protocol – HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[48] ICQ. http://www.icq.com.

[49] InfoMinder. http://www.infominder.com/webminder/index.jsp.

[50] Ives, Z. G., Florescu, D., Friedman, M., Levy, A., and Weld, D. S., "An adaptive query execution system for data integration," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (Philadelphia, PA), May 1999.

[51] J.W.Hunt and M.D.Mcllroy, "An algorithm for efficient file comparison," *Technical Report Computer Science TR#41, Bell Laboratories, Murray Hill, N.J.*, 1995.

[52] Kahler, B. and Risnes, O., "Extending logging for database snapshot refresh," in *Proceedings of the International Conference on Very Large Data Bases*, (Brighton, England), pp. 389–398, September 1987.

[53] Keller, A. M. and Basu, J., "A predicate-based caching scheme for client-server database architectures," in *Proceedings of Parallel and Distributed Information Systems (PDIS)*, September 1994.

[54] Kelly, M. and Seviora, R., "An evaluation of drete on cupid for ops5 matching," in *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, (Detroit, MI), pp. 84–90, August 1989.

[55] Kistler, T. and Marais, J., "Webl - a programming language for the web," in *Proceedings of WWW7 Conference*, (Brisbane, Australia), 1998.

[56] Kistlera, T. and Marais, H., "WebL: a Programming Language for the Web," *http://www.research.digital.com/ SRC/WebL/index.html*, 1998.

[57] Ladin, R., Liskov, B., L.Shrira, and Ghemawat, S., "Programming high availability using lazy replication," *ACM Transactions on Computer Systems, 10,4, pp360-391*, November 1992.

[58] Lee, D. and Chu, W. W., "Semantic caching via query matching for web sources," in *Proceedings of International Conference on Information and Knowledge Management (CIKM)*, 1999.

[59] Lindsay, B., Haas, L., and Mohan, C., "A snapshot differential refresh algorithm," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, (Washington, DC), pp. 53–60, May 1986.

[60] Liu, L., Pu, C., Barga, R., and Zhou, T., "Differential evaluation of continual queries," in *IEEE Proceedings of the 16th International Conference on Distributed Computing Systems*, (Hong Kong), May 1996.

[61] Liu, L., Pu, C., and Han, W., "Xwrap: an xml-enabled wrapper construction system for web information sources," in *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, (San Diego CA), February 2000.

[62] Liu, L., Pu, C., and Tang, W., "Continual queries for internet-scale event-driven information delivery," *IEEE Knowledge and Data Engineering*, 1999. Special Issue on Web Technology.

[63] Liu, L., Pu, C., Tang, W., Buttler, D., Biggs, J., Benninghoff, P., Han, W., and Yu, F., "Cq: a personalized update monitoring toolkit," in *Proceedings of the ACM SIGMOD Conference*, (Seattle, WA), June 1998.

[64] Liu, L., Pu, C., Tang, W., and Han, W., "Conquer: a continual query system for update monitoring in the www," *International Journal of Computer Systems, Science, and Engineering*, 1999. Special Issue on Web Semantics.

[65] Liu, L. and Pu, C., "The diom approach to large-scale interoperable information systems," tech. rep., TR95-16, Department of Computing Science, University of Alberta, Edmonton, Alberta, March 1995.

[66] Liu, L., Pu, C., Barga, R., and Zhou, T., "Differential evaluation of continual queries," Tech. Rep. TR-95-17, Department of Computer Science, University of Alberta, July 1995.

[67] Liu, L., Pu, C., and Han, W., "An XML-enabled data extraction tool for web sources," *International Journal of Information Systems, Special Issue on Data Extraction, Cleaning, and Reconciliation*, 2001.

[68] Liu, L., Pu, C., and Tang, W., "WebCQ: detecting and delivering information changes on the web," *Proceedings of the International Conference on Information and Knowledge Management*, November 2000.

[69] Liu, L., Tang, W., Buttler, D., and pu, C., "Information monitoring on the web: a scalable solution," *World Wide Web Journal*, vol. 5, pp. 263–304, December 2002.

[70] Madden, S. and Franklin, M. J., "Fjording the stream: an architecture for queries over streaming sensor data," in *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, 2002.

[71] Madden, S. R., Shaw, M. A., Hellerstein, J. M., and Raman, V., "Continuously adaptive continuous queries over streams," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (Madison, Wisconsin), May 2002.

[72] Maier, D., *The Theory of Relational Databases*. Computer Science Press, 1983.

[73] McCarthy, D. and Dayal, U., "The architecture of an active database management system," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pp. 215–224, May 1989.

[74] Miranker, D., "Treat: A better match algorithm for ai production systems," in *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pp. 42–47, August 1987.

[75] Model, D. O. http://www.w3.org/DOM/.

[76] Mortice Kern Systems (MKS), "Web integrity." http://www.mks.com/solutions/ebms/, 1997.

[77] Munzner, T. and Burchard, P., "Visualizing the structure of the world wide web in 3d hyperbolic space," in *Proceedings of the 1995 Symposium on Virtual Reality Modeling Language (VRML'95)*, 1995.

[78] NetMind. http://www.netmind.com (discontinued), 2001.

[79] Neuman, B. C., "Scale in Distributed Systems," *IEEE Computer Society Press*, 1994.

[80] Newbery, M., "Kapipo." http://www.vuw.ac.nz/ newbery/Katipo.html, 1996.

[81] Nieh, J. and Lam, M. S., "The design, implementation and evaluation of smart: a scheduler for multimedia applications," in *Proceedings of the 16th ACM Symposium on OS Principles (SOSP)*, 1997.

[82] Opyrchal, L., Astley, M., Auerbach, J., Banavar, G., Strom, R., and D.Sturman, "Exploiting ip multicast in content-based publish-subscribe systems," in *Proceedings of Middleware*, 2000.

[83] Pointcast. http://www.pointcast.com.

[84] Provy, D. and Harrison, J., "A Distributed Internet Cache," *Proceedings of the 20th Australian Computer Science Conference*, Sydney, Feb. 1997.

[85] Rocco, D., Buttler, D., and Liu, L., "Sdiff," *Technical Report, Georgia Tech, College of Computing*, February 2002.

[86] Ronald L. Rivest, "Rfc 1321: The md5 message-digest algorithm." http://www.ietf.org/rfc/rfc1321.txt, April 1992.

[87] Rosenthal, A. and Chakarvarthy, U., "Anatomy of a modular multiple query optimizer," in *The International Conference on Very Large Data Bases*, 1988.

[88] Roussopoulos, N. and Kang, H., "Preliminary design of adms+: a workstation-mainframe integrated architecture fopr database management systems," in *Proceedings of the 12th International Conference on Very Large Data Bases*, (Kyoto, Japan), pp. 355–364, August 1986.

[89] RSS1.0, "Rdf site summary1.0 standard." http://web.resource.org/rss/1.0/.

[90] Satyanarayanan, B., "Scalable, Secure, and Highly Available Distributed File Access," *IEEE Computer, Vol.23, pp.9-21*, May 1990.

[91] Schreier, U., Pirahesh, H., Agrawal, R., and Mohan, C., "Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS," in *Proceedings of the International Conference on Very Large Data Bases*, (Barcelona, Spain), pp. 469–478, September 1991.

[92] Segev, A. and Fang, W., "Currency-based updates to distributed materialized views," in *Proceedings of the 6th International Conference on Data Engineering*, (Los Alamitos), pp. 512–520, February 1990.

[93] Segev, A. and Park, J., "Maintaining materialized views in distributed databases," in *Proceedings of the 5th International Conference on Data Engineering*, (Los Angeles), pp. 262–270, February 1989.

[94] Sellis, T., "Multiple query optimization," *ACM Transactions on Database Systems*, vol. 10, no. 3, 1986.

[95] Sellis, T. and Ghosh, S., "On the multiple-query optimization problem," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 2, no. 3, pp. 262–266, 1990.

[96] SmartBookmarks. http://www.firstfloor.com (discontinued).

[97] Stonebraker, M., Hanson, E., and Hong, C. H., "The design of the Postgres rules systems," in *Proceeding of the International Conference on Data Engineering (ICDE)*, 1987.

[98] Sullivan, M. and Heybey, A., "Tribeca: a system for managing large databases of network traffic," in *Proceedings of the USENIX Annual Technical Conference*, (New Orleans, LA), 1998.

[99] Tang, W., "Personalized update monitoring toolkit using continual queries: System design and implementation," masters thesis, University of Alberta, Edmonton, AB, Canada, June 1998.

[100] Tang, W., Liu, L., and Pu, C., "Trigger grouping: a scalable approach to large scale information monitoring," in *Proceedings The 2nd IEEE International Symposium on Network Computing and Applications (NCA-03)*, (Cambridge, MA), IEEE, April 2003.

[101] Tang, W., Jones, K., Liu, L., and Pu, C., "Bizcq: Using continual queries to cope with changes in business information exchange." Submitted to WWW2004 Conference, November 2003.

[102] Terry, D., Goldberg, D., Nichols, D., and Oki, B., "Continuous queries over append-only databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (San Diego, CA), pp. 321–330, January 1992.

[103] Tewari, R., H.Vin, Dan, A., and Sitaram, D., "Beyond Hierarchies, design considerations for distributed caching on the Internet," *Technical Report TR98-04, Department of Computer Science, University of Texas, Austin*, Feb. 1998.

[104] TracerLock. http://www.tracerlock.com, 2001.

[105] TrackEngine. http://www.trackengine.com.

[106] WatchThatPage. http://www.watchthatpage.com.

[107] WebCatcher. http://www.websense.com/products/about/webcatcher/.

[108] Webclipping.com. http://www.webclipping.com.

[109] WebCQ. http://disl.cc.gatech.edu/WebCQ/.

[110] WebSite-Watcher. http://aignes.com.

[111] WebSpector. http://www.illumix.com/webspector.htm.

[112] WebSprite. http://www.websprite.com (discontinued).

[113] WebWhacker. http://www.webwhacker.com.

[114] Wessels, D. and Claffy, K., "Internet Cache Protocol (ICP)."

[115] Widom, J. and Ceri, S., *Active database systems*. Morgan Kaufmann, 1996.

[116] Williams, S., Abrams, M., Standridge, C., Abdulla, G., and Fox, E. A., "Removal Policies in network caches for World Wide Web documents," *Proceedings of SIGCOMM*, 1996.

[117] WWWFetch. http://www.WWWFetch.com (discontinued), 2000.

[118] XWRAPElite. http://disl.cc.gatech.edu/XWRAPElite.

[119] Yan, T. and Garcia-Molina, H., "The sift information dissemination system," *ACM Transactions on Database Systems (TODS)*, 2000.

[120] Yan, T. W. and Garcia-Molina, H., "SIFT - a tool for wide area information dissemination," in *Proceedings of the 1995 USENIX Technical Conference*, pp. 177–186, 1995.

[121] Zdonik, S., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Carney, D., "Monitoring streams - a new class of data management applications," in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, (Hong Kong, China), 2002.

[122] Zhang, J., Izmailov, R., Reininger, D., and Ott, M., "Web caching framework: analytical models and beyond," *IEEE Workshop on Internet Applications*, 1999.

[123] Zhu, Y. and Shasha, D., "Statstream: statistical monitoring of thousands of data streams in real time," in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, (Hong Kong, China), 2002.

[124] Zhuge, Y., Garcia-Molina, H., Hammer, J., and Widom, J., "View maintenance in a warehousing environment," in *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, (San Jose, CA), May 1995.

[125] Zipf, G. K., "Relative frequency as a determinant of phonetic change," *Reprinted from the Harvard Studies in Classical Philology*, vol. XL, 1929.

VITA

Wei Tang is a Ph.D. candidate at the College of Computing, Georgia Tech. He received his BS degree in Computer Science from Peking University, China in 1996, and his MS degree in Computing Science from the University of Alberta, Canada. Then he spent the first year as a Ph.D. student in Oregon Graduate Institute in the database group before joining the College of Computing at Georgia Institute of Technology in 1999. His research interests include data intensive systems, information retrieval and extraction, information change monitoring, data modelling and visualization, data mining, system scalability and recoverability, and cutting-edge Web technologies. He is involved in various research projects, including Continual Queries, Infosphere (Smart Delivery of Fresh Information), XWRAP (eXtensible Wrapper Generation), AQR (Adaptive Query Routing), and WebCQ (Personalized Information Change Monitoring and Notification Service) . He was one of the key architects and main developer of OpenCQ and WebCQ systems. Besides being a researcher, he is a good badminton player. He also contributes some of his spare time to community services. He has served as a member in the Graduate Student Council at College of Computing (Year 2001) as well as President of the Chinese Friendship Association in Georgia Tech (Year 2000).